Dr. Dobb's SOURCEBOOK
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

SPECIAL ISSUE

Sound, Video, Action!

$ 7.75

TER 1994

MULTIMEDIA PROGRAMMING

FEATURING

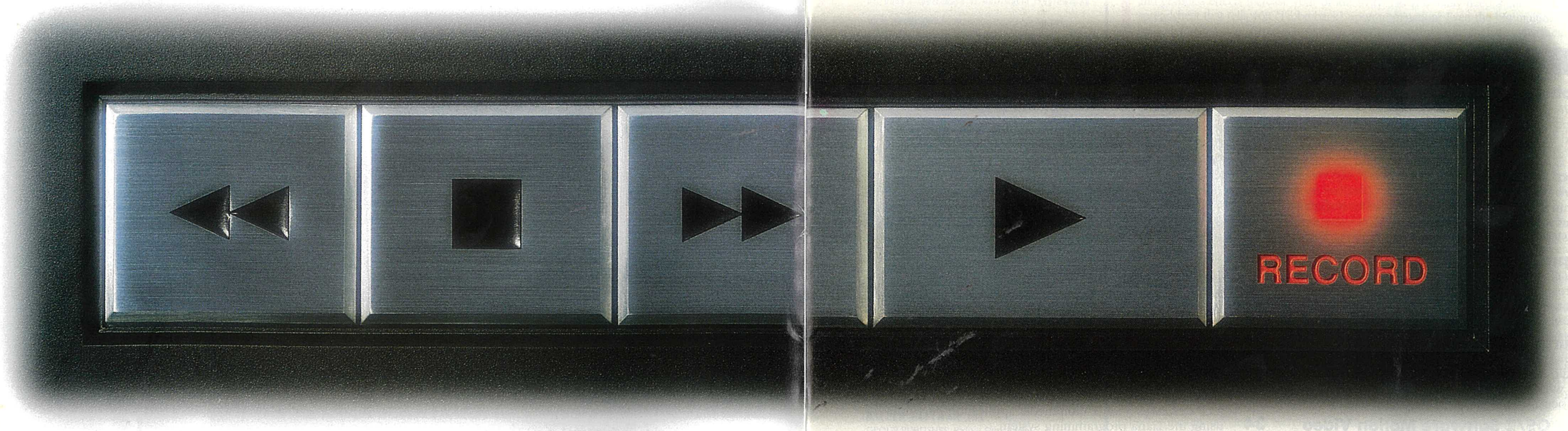
- Digital Video File Formats
- VESA Audio Interface
- Waveform Compression
- Sound Board Roundup
- DSP and Audio Compression

BUILDING YOUR OWN PC SOUND SYSTEM

MORPHING, FULL-MOTION VIDEO, AND MORE!

$4.95 U.S. & Canada
44
0 74470 16562 0

$ 7.75



Intel's video brings the same easy technology concept to your PC.

Introducing Indeo™ technology and the Smart Video Recorder from Intel.

When it comes to putting video technology on your PC, Intel knows exactly which buttons to push.

Intel's new Indeo™ technology is the first video compression technology that lets you capture and compress video in one single step. (Now that's easy.) And the Smart Video Recorder is the first video capture board that takes advantage of Indeo technology. Which means adding the impact of motion video to your business presentations is faster and simpler than you ever imagined.

Recording with the Smart Video Recorder is as simple as pushing a button. And it's fast. It only takes one minute to capture and compress a one-minute video clip. (Other products take up to 15 minutes.) One-step recording also cuts the Smart Video Recorder's disk space requirements—other systems can require up to five times the disk space.

Video for Windows lets you capture, edit and incorporate video into Windows applications.*

A key component of Indeo technology, scalable performance adjusts playback quality based on your system's processing power. And you don't need special hardware for playback. Scalable performance gives you the best playback possible from the computer you have now, and whatever computer you may use in the future.

You can get started right away, too, because Microsoft Video for Windows* and Asymetrix Compel* and MediaBlitz* software programs are included in the Intel Smart Video Recorder box.

To learn more about Indeo technology and the Smart Video Recorder, call 1-800-538-3373, ext. 1154. Or to receive information immediately, call Intel's automated FaxBack* service at 1-800-525-3019 and ask for document #9875. Because with Intel, PC video is all fast forward.

intel®

©1993 Intel Corporation. The Indeo logo is a trademark of Intel Corporation. *Third-party trademarks are the property of their respective holders.

CIRCLE NO. 889 ON READER SERVICE CARD

CONTENTS

Winter 1994
Volume 18, Issue 14
Multimedia Sourcebook

FEATURES

Morphing on Your PC 4
by Scott Anderson
Scott examines the history and mathematics behind morphing and includes three utility programs that let you run full-screen animated morphs on your PC.

Digital Video File Formats 10
by Mark Florence
How can you harness the power of desktop digital video? For starters, by having a grasp of file formats such as Apple's QuickTime and Microsoft's Video for Windows.

Compressing Waveform Audio Files 20
by Neil G. Rowland, Jr.
Neil presents a C++ class library that uses low-level waveform services for signal processing, compressing waveform files to about half the original size.

Multimedia Audio Systems 28
by John Ratcliff
An audio-industry expert examines where we are and where we're going when it comes to multimedia sound systems, focusing on general-MIDI implementations.

Inside OS/2 Software Motion Video 34
by Les Wilson
Les analyzes how OS/2's Multimedia Presentation Manager uses multiple threads for full-motion video synchronization.

Programming the QUANTUMdsp 42
by Charles A. Mirbo
The QUANTUMdsp board lets you download microcode from disk files, thereby redefining the QUANTUMdsp's functionality on-the-fly.

Animation with the Windows GDI 48
by Joe Sam Shirah
Joe Sam presents techniques for *bitblt*s and screen repainting that let you create smoother, faster animations.

The image on the cover is from Masque Publishing's *KaleidoSonics*, a multimedia demo of musical scores and graphics effects.

Writing a Multimedia App in Liana 54
by Jack Krupansky
Jack describes a large multimedia application he wrote using the Liana programming system.

The VESA BIOS Audio Interface 58
by Doug Cody
The VESA BIOS Extension/Audio Interface (VBE/AI) provides a standard software interface for audio, similar to the VGA BIOS for video.

DSP and Audio Compression 63
by Jay B. Reimer
Jay explores the various audio-compression techniques and describes how DSPs can be used for higher performance in multimedia systems.

Build Your Own RS-232 Sound System 70
by Dennis Cronin
Using the tricks Dennis presents here, you can use your PC's RS-232 port to produce serviceable audio from an attached speaker.

Editorial 3
by Michael Floyd

PROGRAMMER'S SERVICES

Source Code Availability

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (California residents add sales tax) to *Dr. Dobb's Journal*, 411 Borel Ave., San Mateo, CA 94402, call 1-415-358-9500, x240, or use your credit card to order by FAX, 1-415-358-9749. Specify issue number and disk format. Code is also available through the DDJ Forum on CompuServe (type GO DDJ), via anonymous FTP from site ftp.mv.com (192.80.84.1) in the /pub/ddj directory, and through M&T Online, a free service accessible via direct dial at 1-415-358-8857 (1200/2400 baud, 8-N-1).

SPECIAL EDITION

Dr. Dobb's JOURNAL — SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

Dr. Dobb's Sourcebook of Multimedia Programming

VICE PRESIDENT-PUBLISHER *Peter Hutchinson*

EDITORIAL
EDITOR-IN-CHIEF *Jonathan Erickson*
MANAGING EDITOR *Tami Zemel*
EXECUTIVE EDITOR *Michael Floyd*
SENIOR TECHNICAL EDITOR *Ray Valdés*
SENIOR PRODUCTION EDITOR *Monica E. Berg*
ASSOCIATE EDITOR *John Dorsey*
ASST. MANAGING EDITOR *Christine de Chutkowski*
ART DIRECTOR *Michael Hollister*
CONTRIBUTING EDITORS *Al Stevens, Tom Genereaux, Andrew Schulman, Ray Duncan, David Betz, Tom Swan*
EDITOR-AT-LARGE *Michael Swaine*
PRODUCTION MANAGER *Amy Shulman Lesovoy*
COVER PHOTOGRAPHER *Michael Carr*

CIRCULATION
DIRECTOR OF CIRCULATION *Jerry Okabe*
CIRCULATION BUSINESS MANAGER *Adela Chau*

ADMINISTRATION
CREDIT MANAGER *Betty Arsene*
ACCOUNTING SUPERVISOR *Renate Kernke*
ACCOUNTS RECEIVABLE *Wendy Ho*

MARKETING/ADVERTISING
ASSOCIATE PUBLISHER *Cynthia C. Sandor*
ADVERTISING COORDINATOR *Sara D. Wood*
SALES/MARKETING ASSOCIATE *Lisa Tarlton*
MARKETING MANAGER *Valerie Dow*
ACCOUNT MANAGERS see page 65

MILLER FREEMAN INC.
GROUP PUBLISHER *Regina Starr Ridley*
GROUP DIRECTOR *Donald Pazour*
CHAIRMAN OF THE BOARD *Graham J.S. Wilson*
PRESIDENT/CEO *Marshall W. Freeman*
SENIOR VICE PRESIDENTS *Thomas L. Kemp, H. Verne Packer, Wini D. Ragus, Stephen G. Stoneburn*
VICE PRESIDENTS *Charles H. Benz, Leonard E. Haas, Vicki L. Masseria, Andrew A. Mickus, Jerry Okabe, Donald Pazour*
CHIEF FINANCIAL OFFICER *Warren Ambrose*

Miller Freeman, Inc.
A publication of M&T Publishing, division of Miller Freeman Inc., member of the United Newspapers Group.

DR. DOBB'S JOURNAL (USPS 307690) is published monthly by Miller Freeman, Inc., 411 Borel Ave., San Mateo, CA 94402-3522; 415-358-9500. Second-class postage paid at San Mateo and at additional entry points.
ARTICLE SUBMISSIONS: Send manuscripts and disk (with article, listings, and letter to the editor) to *DDJ* Submissions, 411 Borel Ave., San Mateo, CA 94022-3522.
***DDJ* ON COMPUSERVE:** Type GO DDJ.
SUBSCRIPTION: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid in U.S. funds drawn on a U.S. bank. Canada and Mexico: $45.00 per year. All other foreign: $70.00 per year.
FOR SUBSCRIPTION QUESTIONS, change of address, and orders, call toll-free **800-456-1215** (U.S. and Canada). For all other countries, call **303-447-9330.** On CompuServe, the customer-service number is 71572,341. Or write, *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188.
POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. **ISSN 1044-789X. GST (Canada) # R124771239.**
FOREIGN NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 30 Montgomery St., Jersey City, NJ 07302; 212-332-7100.
Entire contents copyright © 1994 by Miller Freeman, Inc., unless otherwise noted on specific articles. All rights reserved.

ABP American Business Press — Printed in the USA — The Audit Bureau

EDITORIAL

Why Ask Why?

Is multimedia really a solution without a problem? Can sound, music, full-motion video, and graphics effects increase your productivity? Do you really need a talking spreadsheet that tells you "I'll be back" upon termination? Even the big multimedia-related vendors have trouble answering questions such as these. That's why their response usually sounds more like a "why-ask-why?" commercial for Bud Dry—an answer that tastes good, but is less than filling.

New products—hardware and software—are cropping up faster than developers, users, and even standards committees can keep up with them. Like it or not, multimedia is reaching critical mass.

On the upside for developers, multimedia programming can be downright fun, albeit at times confusing. Which video file format should you use? How can you support all of the available sound cards? What are the best compression schemes? Can a DSP really increase performance without sacrificing flexibility? To help you answer this ever-growing list of questions, we're bringing you *Dr. Dobb's Sourcebook of Multimedia Programming*, a special issue of *Dr. Dobb's Journal* focusing on multimedia software development. Typical of the tricks you'll uncover inside are Scott Anderson's techniques for morphing and Neil Rowland's C++ class library for encapsulating Windows' low-level waveform audio services. Other authors examine digital-video file formats, audio-compression techniques, and programming animation under Windows. You'll also read about the VESA committee's efforts to create a standardized software interface for audio, and—in the spirit of the original *Dr. Dobb's*—a feature on how to roll your own RS-232-based sound system. By the time you've finished reading *Dr. Dobb's Sourcebook of Multimedia Programming*, you'll be on your way to deciding what's hot and what's not when it comes to multimedia applications development.

For instance, among the "what's hot" items is QSound, a virtual-audio technology from Archer Communications (discussed by John Ratcliff) allows left and right panning within a 180-degree "soundscape." The sound technology was first used by the film industry in the movie *Willow*, and has been used by recording artists such as Paula Abdul and Madonna. Game developers have now begun to include QSound in CD-ROM games. Although the QSound algorithms are patented, Creative Technology has licensed the technology from Archer and is providing, free of charge, a QSound API to registered Creative Labs/Technology developers. Sierra Semiconductor has also licensed QSound technology and is currently providing it in their Aria chipset.

Another area that's warming up is speech recognition, and several companies are clamoring to tap into what some experts project will become a one-billion-dollar industry by the end of the decade. At the high end, IBM has ported its RS/6000-based, continuous-speech technology to the 486 and Pentium. Included in its product line is a developer's edition codeveloped with Carnegie Mellon University that sells for $299 and includes a 1000-word vocabulary. Interestingly, both Creative Labs and Sierra Semiconductor have added speech-recognition technology to their product lines. Sierra's Aria Listener is a low-cost speech-recognition engine included as part of its multimedia chipset that also includes the Aria Synthesizer for general-MIDI support. Creative also has a Windows-based speech-recognition engine called VoiceAssist. As with QSound, Creative is providing a VoiceAssist API, which is available for $99.95. Just as interesting is a recent agreement between Borland and Creative Labs to bundle an OWL 2.0-compatible VoiceAssist API in Borland C/C++ 4.0.

Of course, all of this new capability comes at a price. To keep up with emerging technologies, the Multimedia PC Marketing Council has upped the ante on the definition of a "basic" multimedia system. The new MPC level 2 specification defines a multimedia PC as having a 25-MHz 486SX CPU, a minimum of 4 Mbytes of RAM (with 8 Mbytes recommended), a double-speed CD-ROM drive capable of transferring data at 300 Kbytes/second, a 160-Mbyte hard disk, and a VGA+ adapter with a display resolution of 640×480 and 64,000 colors. (For more information, contact the Multimedia PC Marketing Council, 1730 M Street NW, Suite 707, Washington, DC 20036.)

Given this relatively high cost for entry, the technical hoops to make it work, and the lack of clear productivity gains, you have to wonder if multimedia is worth the effort.

Recall that in the early days of the PC color added little to the immediate productivity of users, yet the CGA quickly became standard fare. Obviously, if history is any gauge, appearances are indeed important. Now, sight and sound allow multimedia developers to involve users in ways not readily possible before. What you see *and* hear is what you get.

Michael Floyd
Executive Editor

Morphing on Your PC

An easy-to-use algorithm for animation and morphing

Scott Anderson

Morphing has become almost as ubiquitous as bubble-gum under theater seats. As the July 1993 issue of *Dr. Dobb's Journal* illustrated, from autos turning into tigers to bulls into bears, we see examples of morphing everywhere on television and in movies. While the effect is compelling, morphing is useful beyond the magical transformations we've grown used to seeing.

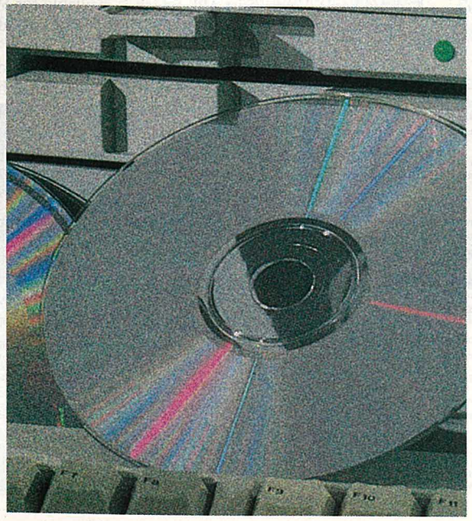
This article examines the history and mathematics behind morphing, and includes three utility programs that allow you to run full-screen animated morphs on your PC.

Warping, Tweening, and Dissolving

Morphing is a blend of three separate algorithms: warping, tweening, and dissolving. *Warping* is the mathematical trick of stretching and squashing an image as if it were painted on rubber. This article discusses an implementation of one of the latest warping routines, based on the method used by Pacific Data Images (PDI) to make the Michael Jackson video "Black or White." The algorithm uses *lines* to control the warping. Lines make the warping specification much easier than using points, giving the artist a break.

Scott is the president of Wild Duck, a software publishing and development company in Rohnert Park, California. He is also the author of the animation program Fantavision, and the recently released book Morphing Magic *from SAMS. He can be reached at 73710,1055 on CompuServe.*

Tweening is short for in-betweening, the interpolation of two images to yield a smooth-flowing animation. Tweening is typically done with points, lines, or polygons. Since this warping algorithm is line oriented, tweening fits right in. By tweening the position of the control lines, you can smoothly warp an image around. With warping and tweening alone, you can create photorealistic animations from single photographs. And it's simple to do. (For more details, see my book *Morphing Magic*, SAMS, 1993.)



Dissolving, or cross-dissolving, is Hollywood-speak for fading out one scene while fading in the next. In the middle, you have a double exposure. This powerful effect was used in the early Wolfman movies. While poor Lon Chaney, Jr. stuck his head in a vice, makeup people swarmed about, adding hair and putty. After each little change in makeup was completed, another few frames of film were squeezed off. Each short take was cross-dissolved with the previous one to complete the illusion.

When you put all three of these effects together, you end up with morphing. Here's how it works. Let's say you want to morph yourself into a tiger. On the first frame of your self-portrait, you mark the key areas with lines. You might place a control line from one eye to the other, down the nose, and across the lips. These three lines capture the most important aspects of a face. Then, on the tiger's ferocious countenance, you would draw control lines on the same features: eyes, nose, and mouth. That's enough information for the computer to morph the two. To create a single morph in the middle, the control lines are tweened halfway and then the images are warped to follow the new lines. The algorithm warps your face midway to the tiger, and warps the tiger midway to you. This is the fifty-fifty interpolation between the two sets of control lines—the central tween. It is also simply the average of the two sets.

After creating the two warped images, the routine cross-dissolves them, finally producing the morph. If you use more tweens, say ten interpolations between you and the tiger, you can make a smoothly animated sequence of you turning into the tiger. The first frame uses 10 percent of the tiger mixed with 90 percent of you. The second frame has 20 percent of the tiger with 80 percent of you. By the ninth frame, there is 90 percent tiger, and you have faded out to 10 percent.

Warping Algorithms

There are as many ways to warp as there are sort routines. The method employed here is a variation on the PDI line method. As I mentioned before, this approach is friendly to the artist. Other routines use dots to warp the image. These changes are local and act only over a short distance, so you need hundreds of them. Misplacing some of these dots and ruining the warp isn't at all difficult. With the new method, each line represents the string of points that compose it, so a few short lines can stand in for hundreds of points. As you move a control line, the pixels around it are pulled as well.

One of the interesting peculiarities of this algorithm is the global nature of the warping. A single line can specify any combination of scaling, translation, and rotation. This can be a nice effect in itself. Just draw, say, a vertical line, rotate it 90 degrees, and the image will also be rotated. When there are more lines (as there usually are), they will each compete for influence, but the effect will still be global. The downside is that it takes a little longer to perform the warping.

In the algorithm presented here, the influence of the line falls off as $1/distance^2$. This is just like the influence of gravity, but you can select any variation on this that you desire. This formulation was chosen based largely on speed considerations. A mathematical discussion of the warping algorithm is in the text box entitled, "Warping with Control Lines." The basic warping algorithm is as follows:

Morphing is a blend of three separate algorithms: warping, tweening, and dissolving

1. Find the distance between a pixel and the source line by dropping the perpendicular *d* (see Figure 1).
2. Find the fractional distance *f* along the source line to the perpendicular. The fraction is normalized to go from 0 to 1.
3. Move the point to a spot that is the same distance from, and fraction of, the *target* line.

This algorithm is carried out by two routines, *sumLines* and *getSourceLoc*; see Listing One, page 75. In *getSourceLoc*, the warped pixel is determined for each line, while *sumLines* adds up the influence of each line. *GetSourceLoc* performs the mathematics given in the text box entitled, "Warping with Control Lines." *SumLines* uses the distance returned by *getSourceLoc* to determine the weight contributed by each control line. In a loop through the lines, *SumLines* calculates the individual weights, as shown in Figure 2. You can experiment with the weight calculation (.001 is added to avoid division-by-zero errors). Try equations that depend on the length of the line, or inversely on the cube of the distance. The method used here is simple and fast, but don't be afraid to try something new. You never know what will happen!

Tweening Algorithm

The second piece of the morphing trio is tweening. Compared with warping, tweening is a piece of cake. In the simplest case of linear tweening, all you do is interpolate. *Linear* interpolation is just finding a point on the *line* connecting two others. Although there are dozens of ways to interpolate, for our purposes linear tweening is just fine. All we do is take the lines describing the source image and tween them into the lines describing the target. For each intermediate tween, we warp the images accordingly.

For speed, divide the distance between the points into as many steps as you desire, say ten. You will have an X and a Y component for the length of these segments, called "deltas." For a regular division of the line, all these deltas will be equal to one-tenth of the line distance, so you only need to calculate it once and then simply add it to the starting point. After ten additions of this delta, you will have arrived at the target point. The deltas are given by the simple equations in Figure 3. Adding a delta in this fashion is fast, but care must be taken to avoid round-off errors.

Dissolving Algorithm

Now we are at the third and final part of morphing, the dissolve. This is where you combine the two images you are warping and tweening. Mix any two colors just the way you might expect. The color midway between two colors is given by the formulas in Figure 4. On a 24-bit color system, the color you get from this calculation will always be another color you can display. Not so with color-mapped graphics cards. You can

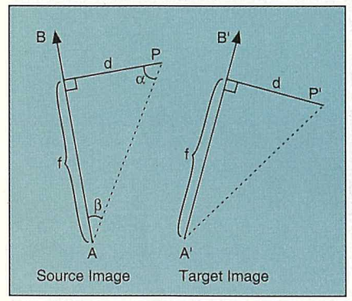


Figure 1: *A given point* ***P*** *has a relationship to the line* ***AB*** *that gets mapped to the warping line* ***A'B'****.*

```
distance = getSourceLoc(&orig,origline,warp,warpline);
weight   = 1/(.001+distance*distance);
deltaSumX      += (orig.x - warp->x) * weight;
deltaSumY      += (orig.y - warp->y) * weight;
```

Figure 2: *Calculating individual weights.*

```
DeltaX = (TargetX - SourceX) / (Tweens + 1)
DeltaY = (TargetY - SourceY) / (Tweens + 1)
```

Figure 3: *Equations to calculate the interpolation deltas.*

```
NewRed = (SourceRed + TargetRed) / 2
NewGreen = (SourceGreen + TargetGreen) / 2
NewBlue = (SourceBlue + TargetBlue) / 2
```

Figure 4: *Calculating the color midway between two colors.*

```
MORPH <File1> <File2> [<Steps> [<OutFile>]]

<File1> is the name of the source PCX file to morph.
<File2> is the name of the target PCX file.
<Steps> is the optional number of output files.

If you don't specify the number of steps in your sequence, the program
produces one in-between morph. You must specify <Steps> if you want to
output files.

<OutFile> is the root name of the optional output files. The value of <Steps>
determines the number of files to output. The step number is appended to the
root name, so the <OutFile> name must be six characters or less.
```

Figure 5: *Command-line syntax for calling MORPH.*

easily derive a color that isn't in the puny list of displayable colors called the "palette." The best most VGA cards can offer is 256 colors, which is pretty sad for this application.

To fix this you need to calculate all the colors as if you had 24 bits to play with. Then, you need to create a new palette that uses the 256 most popular colors. The rest of the colors must be forced into the closest color available from those 256. This process is referred to as "palette mapping."

So dissolving, which should be the smallest part of this whole algorithm, suddenly swells into an unsightly carbuncle on this otherwise straightforward code. Those are the basics of morphing. There are many different algorithms for performing this terrific effect, but this should serve as a good jumping-off point for further experimentation.

The Morphing Programs

So that you can experiment, I've included three programs: MORPH.C (Listing Three, page 77), LOAD.C (Listing Four, page 78), and FIX.C (Listing Five, page 78). The header containing the *#define*s for all three of these utilities is in Listing Two, page 76. MORPH gets the user input and creates the morphing sequence, FIX finds a good palette and LOAD plays the sequence back from memory. All the programs work with PCX files having 256 colors and 320×200 resolution. You can capture your own PCX files with any screen-capture program, or you can find GIF images on CompuServe and convert them with a program such as ZSoft's Paint Shop. Remember to convert the number of colors to 256. If you have access to a scanner, then your troubles are over. You can scan in your own photos and magazine clips and go from there.

When specifying a PCX file, you don't need to type in the file extension. The programs automatically append ".PCX" to the end of each filename, saving you the trouble.

Some of these programs produce a numbered sequence of output files. These are also PCX files. The maximum sequence size is 99. The sequence number, from 1 to 99, is appended to the root name of the file sequence. This takes two characters away from the already-tiny DOS allotment of eight. Therefore, the <OutFile> name must be six characters or less.

Figure 5 shows the syntax used on the command line to call the MORPH program. MORPH first reads the two input files: the source and target PCX images. Then it looks on the disk for any existing control lines that might be associated with the source or target image. If the user okays it, these line files are loaded in. Otherwise, the user creates a new set of control lines from scratch, and these are saved to the disk. MORPH metamorphoses between the two in the given number of steps, to make a sequence of files. The files are numbered and can be displayed by the LOAD program. As an example, issuing the command line MORPH JIM BOB will cause MORPH to load the files JIM.PCX and BOB.PCX. It will create the middle tween and display it. No files are created. However, MORPH JIM BOB 7 will load the files JIM.PCX and BOB.PCX, create a sequence of seven frames, and display them. No files are created. Finally, MORPH JIM BOB 7 JIMBOB will also load JIM.PCX and BOB.PCX and create a sequence of seven frames. Each frame is displayed while it is saved to the disk. The files will be named JIMBOB1.PCX, JIMBOB2.PCX, and so on.

```
LOAD <File>

<File> is the root name of a PCX
sequence to display.
```

Figure 6: *Using the LOAD program.*

Figure 6 shows the syntax used to call the LOAD program. LOAD reads a sequence of PCX files indicated by the given name. It reads as many as it can fit into memory, then it displays them sequentially. You can control the playback rate by pressing a number from 1 (slow) to 9 (fast). To single-step through the pictures, press Spacebar. To continue playback, press Enter. To quit, press Esc. The number of images that can be animated depends on how much free RAM you have. Get rid of your TSRs and use the new memory features of DOS 6.0. The best you can do in 640K is about eight images.

The command-line syntax used to call FIX is in Figure 7. FIX takes a sequence created by MORPH and forces each picture in the sequence to have the same palette. FIX then writes the sequence out under the new name. The command FIX JIM FXJIM reads the sequence JIM1.PCX, JIM2.PCX, and so on, outputting the same number of files named FXJIM1.PCX, FXJIM2.PCX, and so. Also, FIX JIM JIM reads the JIM sequence and writes over the original files. Be careful! Make sure you don't need the originals before you do this.

Compiling

These programs were all compiled with Microsoft C/C++ 7.0. If you didn't include the graphics library when you installed your compiler, you will need to link that library in at compile time. I've

A History of Warping

Warping's history reaches back to the '60s space program, when NASA was snapping shots of the earth like an eager tourist. But when it came time to put all the pictures together, NASA discovered that nothing quite lined up correctly because of the different altitudes, angles, times, and optics for each shot. Consequently, researchers at NASA developed algorithms that treated the digitized data as points on a polynomial surface that could be stretched to fit a set of reference points. They called their efforts *image registration.* Those marvelous Landsat pictures were thus stretched and pulled into a big quilt of pictures blanketing the globe. This was a great start for a little warping routine.

Warping was dusted off in the mid-seventies, when Viking 2 landed on Mars. Unfortunately, Viking came to rest at an angle, with its camera pointing down. The landscape it depicted looked like the Big Valley, with the horizon strongly curved. All the images of Mars were pushed through a warping algorithm that corrected for the odd optics before the public ever saw them.

Warping has found its way into the medical world, too. In a procedure called "digital subtraction angiography," an X-ray of a patient is taken before and after the injection of a dye into the area of interest. By subtracting the first image from the second, everything is removed but the dyed arteries. This uncluttered image is of great value to the diagnostician. Unfortunately, if the patient moves, the effect is wrecked. And, except for tree surgeons, moving patients are the norm. As you might have guessed, warping is the perfect tool to ensure registration.

— S.A.

Warping with Control Lines

Given point **P** in the source image, you want to deduce point **P'** in the target image (see Figure 1). The distance d is the projection of vector **AP** on the perpendicular, which yields $|\mathbf{AP}| \cos \alpha$, where $|\mathbf{AP}|$ represents the magnitude of the vector. The dot product of vector **AP** with the perpendicular to **AB** (denoted $\perp\mathbf{AB}$) is defined in Example 1(a), where $|\mathbf{AP}|$ and $|\mathbf{AB}|$ are the magnitudes, and the subscripted variables are the x and y components of the vectors. You can see that this solution in terms of components, $(AP_x * \perp AB_x + AP_y * \perp AB_y)$, doesn't involve any trigonometry. Example 1(b) solves for d.

The magnitude of the line itself, $|\mathbf{AB}|$ is used instead of the magnitude of the perpendicular, $|\perp\mathbf{AB}|$. The reason is that these two are equal, and you can reuse this number. The vector dot product allows you to calculate the desired values without computing any angles or using any transcendental functions. That makes it both simple and fast. From Figure 1, note that the distance represented by f is the projection of **AP** on **AB** itself, $|\mathbf{AP}| \cos \beta$, which is the quantity: $(\mathbf{AP} \cdot \mathbf{AB})/|\mathbf{AB}|$. We want the fractional part of line **AB**, so we divide by the length of **AB** again as shown in Example 1(c).

Examples 1(b) and 1(c) represent a translation into a new-scaled orthogonal coordinate system based on d and f, instead of x and y. Now you are ready to transfer these two important relationships over to the new line, **A'B'**. The fractional part, the "f-axis," is simply the fractional part of the *new* line, **A'B'**: f***A'B'** Next, apply the distance to the "d-axis," which is perpendicular to the new line. The unit d-vector is $(\perp\mathbf{A'B'})/|\mathbf{A'B'}|$. So, with the new origin at **A'**, the source pixel **P** (represented as a vector from the origin) is transformed into the destination pixel **P'**; see Example 1(d). This algorithm uses the relationship a point has with the original line and applies it to the new line. To be of real use, however, you need more lines. They all have to compete for pixels. As mentioned in the text, this implementation uses a weighting proportional to one over the distance squared.

In the calculation for two control lines, the distance to the point is computed from each line, and a new pixel is calculated as before. But this time there are two lines and therefore two warped pixels, so some further work is needed. As shown in Figure 8, from the original point **P** to the new points $\mathbf{P}_1'$ and $\mathbf{P}_2'$ there are two displacements, $\mathbf{D}_1$ and $\mathbf{D}_2$; see Example 1(e). The routine calculates the weighted average of the two displacements to arrive at the final position, **P'**; see Example 1(f). If there are three lines, there are three displacements to average, and so on. Finally, the general equation for calculating with n lines is shown in Example 1(g).

— S.A.

(a) $\mathbf{AP} \cdot \perp\mathbf{AB} = |\mathbf{AP}| * |\perp\mathbf{AB}| \cos\alpha$

$= (AP_x * \perp AB_x + AP_y * \perp AB_y)$

(b) $d = |\mathbf{AP}| \dfrac{\mathbf{AP} \cdot \perp\mathbf{AB}}{|\mathbf{AB}|}$

(c) $f = \dfrac{\mathbf{AP} \cdot \mathbf{AB}}{|\mathbf{AB}|^2}$

(d) $\mathbf{P'} = \mathbf{A'} + f * \mathbf{A'B'} + d * \dfrac{\perp\mathbf{A'B'}}{|\mathbf{A'B'}|}$

(e) $D_1 = P_1' - P$

$D_2 = P_2' - P$

(f) $P' = P + \dfrac{D_1 W_1 + D_2 W_2}{W_1 + W_2}$

(g) $P' = P + \dfrac{\sum_{i=1}^{n} D_i W_i}{\sum_{i=1}^{n} W_i}$

Example 1: *(a) Dot product of vector **AP** with the perpendicular to **AB**; (b) solving for d; (c) calculating the fractional part of line **AB**; (d) transforming the source pixel **P** into the destination pixel **P'**; (e) determining the displacements $\mathbf{D}_1$ and $\mathbf{D}_2$; (f) weighted average of the two displacements; (g) solving for the general case using n lines.*

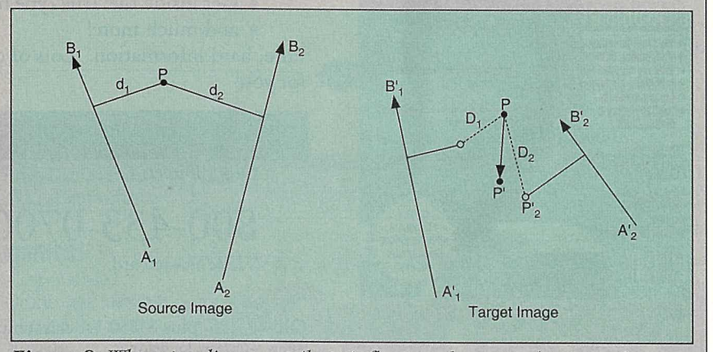


Figure 8: *Where two lines contribute influence, the warped point is the weighted average of the displacements $\mathbf{D}_1$ and $\mathbf{D}_2$.*

Lines make the warping specification much easier than using points, giving the artist a break

```
FIX <InFile> <OutFile>

<InFile> is the root name of the PCX sequence to fix.
<OutFile> is the root name of the fixed PCX sequence.
```

Figure 7: *Command-line syntax used to invoke the FIX program.*

included a makefile, which is available electronically; see "Availability," page 2. The makefile is set up assuming that you *haven't* installed the graphics library. If you *have* installed the graphics library, then you'll have to modify this makefile accordingly.

To make the programs work with other C compilers, you will probably need to make some changes to the non-ANSI library calls. In Microsoft C, these library calls are prefixed with the underscore character. The non-ANSI library calls are all in IO.C (available electronically) and LINECALC.C (Listing One).

Conclusion

The most obvious upgrade to this program is the ability to use extended RAM, so you can hold more frames in memory. If you have access to some of the new software on the market that attempts to leap the 640K barrier, use it. If you can work with smaller images, say, quarter-screen, 160×100 resolution, you should be able to quadruple the number of frames that can be held in memory, not to mention speeding up the animation. The display-picture routines you need to change are in Listing Five. Currently, they slam a continuous string of bytes into memory. For smaller screens, you will need to move the image a line at a time.

If you have a true-color card, you don't need the agonizing color-collapsing routines from Listing Six (page 79). Just yank them all out. You will need to save the files in 24-bit color mode, which can present some data problems, so look at the drivers and documentation for your particular graphics card.

With these programs, you can create some amazing morphs. With a video capture board and a computer capable of displaying 30 frames per second, you can output to videotape. Now you can play with some of the same toys they have at Industrial Light and Magic and Pacific Data Images. Have fun!

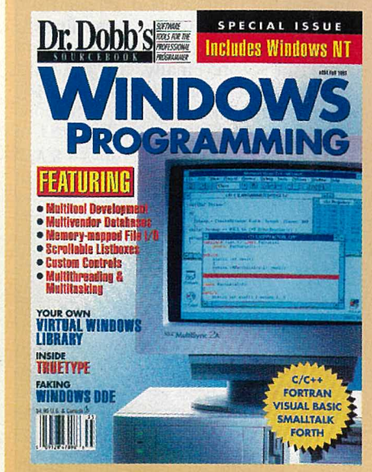
DDJ

(Listings begin on page 75.)

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

Dr. Dobb's Journal Special Issue

Sourcebook of Windows Programming



You demand expert, unique information on developing for Windows. The ***Dr. Dobb's Sourcebook of Windows Programming*** gives you that singular perspective only *Dr. Dobb's Journal* provides. Find out how to:

- Develop multivendor Windows databases
- Write custom controls
- Create horizontally scrollable list boxes
- Get inside the TrueType format
- and much more!

Pure, hard information. Lots of code. Put this special issue to work for you!

Order Direct:

800-433-0700

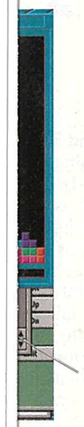
VISA/Mastercard

Mail check to:
Dr. Dobb's Journal
PO Box 105843
Atlanta, GA 30348-5843

Only $4.95, plus $1.50 1st class shipping. Supplies limited.
Allow 4-6 weeks for delivery.

TM

UnixWare users! Get a Head Start on Your New Year's Resolution



Comfortable 'look and feel' for both keyboard and mouse users.

With scalable window capability provided by Adobe Type Manager,™ you can get more DOS windows on the screen than ever before.

Any DESQview/X window can be made into an icon at any time.

Online help is two keystrokes away.

Customization

…le menus, graphical desktop, keystroke …nline help, it is very easy to tailor the your …eeds. In addition, you can choose from two …: OSF/Motif and OPEN LOOK.

…le Companion Programs

…companion programs. Application Manager …rograms. File Manager for managing local …or for creating icons.

…C a better place to work, whatever software …

	5-1/4	3-1/2	Each	Totals
			$275.00	
…etwork Mgr. (TCP/IP)			$200.00	
…286 PCs)			$99.95	
…(for 386 and 486 PCs)			$149.95	
Shipping & Handling $10 (Canada & USA only)				
California Residents add 8.25%				
Grand Total				

…Visa …MasterCard Expires ___/___
Card #
Name
Address
City State Zip

Call (800) 354-3222 ext. 1D1 for fastest service Please allow 3 weeks for delivery

…DESQview/X has the memory management services of advanced operating systems—thanks to the built-in Rational Systems DOS4GX DOS extender (16 and 32 bit), shared libraries, dynamic link libraries (DLLs) and virtual memory.

Superior DOS Memory Management

DESQview/X comes with our award-winning, best-selling expanded memory manager, QEMM, and the highly acclaimed Quarterdeck Manifest to ensure that your PC's memory is always at its optimum and that your DOS programs have the absolute maximum memory available.

Quarterdeck

Quarterdeck Office Systems, 150 Pico Boulevard, Santa Monica, CA 90405 (310) 392-9851 Fax (310) 314-4219
Quarterdeck International Ltd., B.I.M. House, Crofton Terrace, Dun Laoghaire Co. Dublin, Ireland Tel.(353) (1) 284-1444 Fax: (353) (1) 284-4380

©1993 Quarterdeck Office Systems. Trademarks are property of their respective owners.

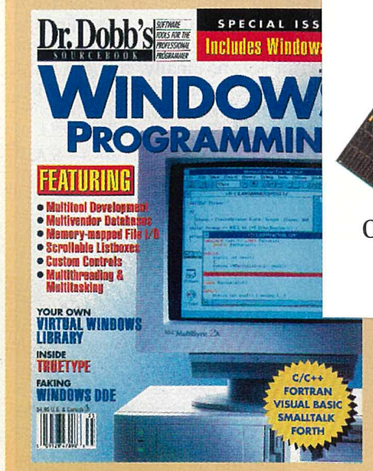
CIRCLE NO. 501 ON READER SERVICE CARD

Lines make the warping specification much easier than using points, giving the artist a break

included a makefile, which is availa
electronically; see "Availability," pag
The makefile is set up assuming t
you *haven't* installed the graphics
brary. If you *have* installed the gra
ics library, then you'll have to mo
this makefile accordingly.

To make the programs work w
other C compilers, you will proba
need to make some changes to
non-ANSI library calls. In Microsof
these library calls are prefixed with

Dr. Dobb's Journal Specia
Sourcebo



Give your UnixWare PC the color and resolution of a graphics workstation with Quarterdeck's Hyper-X™ software and your graphics accelerator card

Hyper-X high performance graphics accelerator software and your graphics board can take you from 640 x 480 to 1280 x 1024 or even 1600 x 1200 in seconds

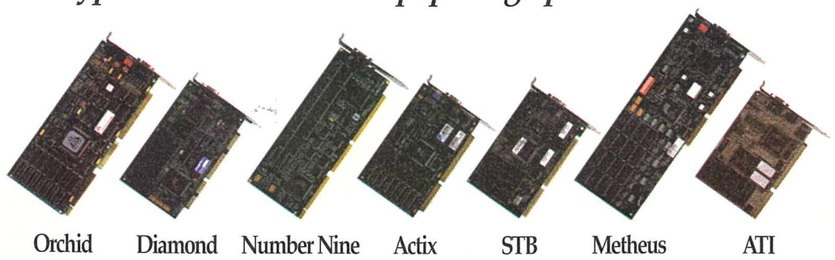
Just in time for the Holidays! Now, for only few hundred dollars, you can transform your ordinary Intel-based UnixWare PC into a machine whose display rivals Unix graphics workstations!

New from Quarterdeck, the leaders in memory management, DOS/Windows multitasking and X-Windows/DOS interoperability, Hyper-X takes the emerging client-server standard to the next level, bringing the full graphics power of a dedicated, expensive workstation to UnixWare users on the network.

Before: 640 x 480 16-color VGA Display (even with S3 or ATI board)

Hyper-X works with your choice of graphics accelerator board to provide up to a ten-fold performance increase, along with greater display resolution and color capability to UnixWare.

Hyper-X works with these popular graphics accelerators

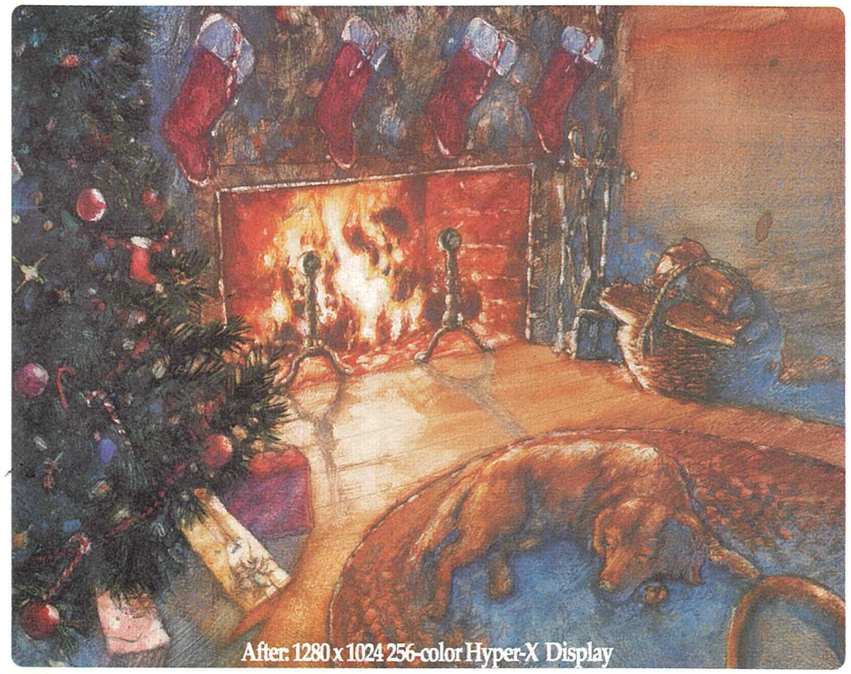


Orchid Diamond Number Nine Actix STB Metheus ATI

© 1993, Quarterdeck Office Systems. Trademarks are property of their respective owners.

Your dream of speed and power comes true

It may seem incredible that your trusty PC could handle nearly four times the screen resolution, plus more colors and still outperform a standard VGA or SVGA display by up to ten times, but the answer lies in the design of your graphics accelerator. It has its own specialized processor, which takes all the time-consuming display tasks from your CPU and runs them far more efficiently. The result is an overall increase in performance and display quality that will take your breath away.



After: 1280 x 1024 256-color Hyper-X Display

Hyper-X and your compatible graphics accelerator card make your UnixWare PC into the fast, powerful machine you always dreamed it could be.

Hyper-X is expected to ship in time for Christmas, 1993.

Quarterdeck

DESQview/X has the memory management services of advanced operating systems—thanks to the built-in Rational Systems DOS4GX DOS extender (16 and 32 bit), shared libraries, dynamic link libraries (DLLs) and virtual memory.

Superior DOS Memory Management

DESQview/X comes with our award-winning, best-selling expanded memory manager, QEMM, and the highly acclaimed Quarterdeck Manifest to ensure that your PC's memory is always at its optimum and that your DOS programs have the absolute maximum memory available.

TM

W/X

MS Windows
PC—or any

Comfortable 'look and feel' for both keyboard and mouse users.

With scalable window capability provided by Adobe Type Manager,™ you can get more DOS windows on the screen than ever before.

Any DESQview/X window can be made into an icon at any time.

Online help is two keystrokes away.

Customization

le menus, graphical desktop, keystroke
nline help, it is very easy to tailor the your
eeds. In addition, you can choose from two
: OSF/Motif and OPEN LOOK.

le Companion Programs

companion programs. Application Manager
ograms. File Manager for managing local
r for creating icons.

C a better place to work, whatever software

	5-1/4	3-1/2	Each	Totals
			$275.00	
etwork Mgr. (TCP/IP)			$200.00	
286 PCs)			$99.95	
(for 386 and 486 PCs)			$149.95	
Shipping & Handling $10 (Canada & USA only)				
California Residents add 8.25%				
			Grand Total	

Visa MasterCard Expires ___/___
Card #
Name
Address
City State Zip

Call (800) 354-3222 ext. 1D1 for fastest service
Please allow 3 weeks for delivery

Quarterdeck

Quarterdeck Office Systems, 150 Pico Boulevard, Santa Monica, CA 90405 (310) 392-9851 Fax (310) 314-4219
Quarterdeck International Ltd., B.I.M. House, Crofton Terrace, Dun Laoghaire Co. Dublin, Ireland Tel.(353) (1) 284-1444 Fax: (353) (1) 284-4380

©1993 Quarterdeck Office Systems. Trademarks are property of their respective owners.

CIRCLE NO. 501 ON READER SERVICE CARD

Order Direct:

800-433-0700

VISA/Mastercard

Mail check to:
Dr. Dobb's Journal
PO Box 105843
Atlanta, GA 30348-5843

Only $4.95, plus $1.50 1st class shipping. Supplies limited.
Allow 4-6 weeks for delivery.

Lines make the warping specification much easier than using points, giving the artist a break

included a makefile, which is avai
electronically; see "Availability," pa
The makefile is set up assuming
you *haven't* installed the graphi
brary. If you *have* installed the gr
ics library, then you'll have to m
this makefile accordingly.

To make the programs work
other C compilers, you will prob
need to make some changes t
non-ANSI library calls. In Microso
these library calls are prefixed wit

Dr. Dobb's Journal Speci
Sourceb

Dr. Dobb's SOURCEBOOK — SPECIAL IS — Includes Windo
WINDOW PROGRAMMI
FEATURING
- Multitool Development
- Multivendor Database
- Memory-mapped File I/O
- Scrollable Listboxes
- Custom Controls
- Multithreading & Multitasking

YOUR OWN VIRTUAL WINDOWS LIBRARY
INSIDE TRUETYPE
FAKING WINDOWS DDE
C/C++ FORTRAN VISUAL BASIC SMALLTALK FORTH

Order Direct:
800-433-0700
VISA/Mastercard

Mail check to:
Dr. Dobb's Journal
PO Box 105843
Atlanta, GA 30348-5843

Only $4.95, plus $1.50 1st class shipping. Supplies limited.
Allow 4-6 weeks for delivery.

Get to know the growing family of Quarterdeck performance-enhancing software

Hyper-X

The easiest way to give UnixWare PCs graphics workstation capability. Combine Hyper-X software with the accelerator card of your choice and you can have the high resolution (up to 1600 x 1200 with 256 colors) display capability of a dedicated workstation. Plus, your 386/486 PC will run faster. Hyper-X is compatible with all UnixWare applications. And like all Quarterdeck products, Hyper-X offers advanced installation, configuration and diagnostics capability.

QEMM 7

The acknowledged leader in memory management software gets even better in this, the seventh release. It gives you up to 96K more than other memory managers provide for programs, TSRs and utilities. More memory allows your programs to run faster, more smoothly, and with less chance of data loss. It also means you have more freedom and flexibility to accommodate network drivers, plus productivity or multimedia utilities, TSRs and drivers.

DESQview

Our pioneering multi-windowing multitasking environment gets better every year. It's the easy, fast way to run DOS and MS-Windows applications side-by-side. QEMM is included to manage memory for optimum performance.

DESQview/X

Multiplatform interoperability becomes a reality with DESQview/X, the environment that allows PCs to remotely run X Window programs on their network's workstations. PCs can also remotely run programs on other PCs, so the slowest PC on the network can take advantage of the fastest machines available to get their work done as quickly as possible.

More to come...

Quarterdeck is working on a range of productivity solutions for today's and tomorrow's enterprise computing challenges. Watch for new developments.

Call for Special Introductory Offer on Hyper-X or for a brochure with more informati
(800) 354-3222 Ext. 8G8

Quarterdeck

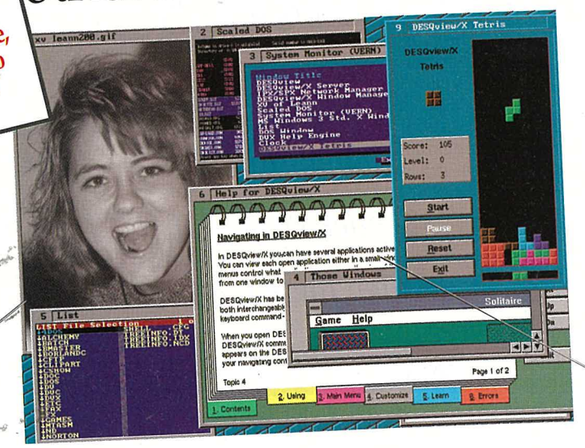
Quarterdeck Office Systems, 150 Pico Boulevard, Santa Monica, CA 90405
(310) 392-9851 Fax: (310) 314-4219
Quarterdeck International Ltd., B.I.M. House, Crofton Terrace, Dun Laoghaire Co. Dublin, Ireland
Tel. (353) (1) 284-1444 Fax: (353) (1) 284-4380

TM

DESQview/X

The simplest way to integrate DOS, MS Windows and X Window software on your PC—or any other PC on a network.

New Version 1.2 includes enhanced network support: PC-NFS, Hewlett-Packard LAN Manager, Microsoft LAN Manager, Beame & Whiteside, Wollongong—in addition to PC/TCP and Novell LAN Workplace for DOS



DOS and Windows applications compatibility.

Proven memory management and diagnostics with both Manifest and QEMM.

DESQview/X has EGA, VGA, 8514A, Super VGA and DGIS high resolution graphics support.

Comfortable 'look and feel' for both keyboard and mouse users.

With scalable window capability provided by Adobe Type Manager,™ you can get more DOS windows on the screen than ever before.

Any DESQview/X window can be made into an icon at any time.

Online help is two keystrokes away.

Multitasking Power

DESQview is the recognized pioneer in DOS multitasking and is at the very heart of DESQview/X—giving you proven high performance multitasking of DOS and Microsoft Windows programs side-by-side.

X Window System Graphics

The X Window System gives DESQview/X its graphical interface And its ability to access DOS text and MS Windows programs running remotely on your network. Add the optional TCP/IP Network Manager and it gives you the ability to access remote X Window programs on workstations such as an IBM RS/6000 or a SUN SPARCstation. Or vice versa.

Adobe Type Manager

The accepted standard in scalable font technology is built in so that X Window System programs can use scalable fonts. And it also gives you the ability to view DOS text programs in scaled windows.

Program Memory Management

DESQview/X has the memory management services of advanced operating systems—thanks to the built-in Rational Systems DOS4GX DOS extender (16 and 32 bit), shared libraries, dynamic link libraries (DLLs) and virtual memory.

Superior DOS Memory Management

DESQview/X comes with our award-winning, best-selling expanded memory manager, QEMM, and the highly acclaimed Quarterdeck Manifest to ensure that your PC's memory is always at its optimum and that your DOS programs have the absolute maximum memory available.

Customization

With DESQview/X's customizable menus, graphical desktop, keystroke macros, mark and transfer and online help, it is very easy to tailor the your own PC to your exact working needs. In addition, you can choose from two optional 'look and feel' packages: OSF/Motif and OPEN LOOK.

Includes Valuable Companion Programs

DESQview/X comes with three companion programs. Application Manager for launching and organizing programs. File Manager for managing local and remote files. And Icon Editor for creating icons.

Quarterdeck makes your PC a better place to work, whatever software you prefer to use.

Yes! Please Send:

Qty	Product	5-1/4	3-1/2	Each	Totals
	DESQview/X			$275.00	
	DESQview/X Network Mgr. (TCP/IP)			$200.00	
	DESQview (for 286 PCs)			$99.95	
	DESQview 386 (for 386 and 486 PCs)			$149.95	
				Shipping & Handling $10 (Canada & USA only)	
				California Residents add 8.25%	
				Grand Total	

Payment method ☐Visa ☐MasterCard Expires ___/___
Card # _____
Name _____
Address _____
City _____ State ____ Zip ____

Call (800) 354-3222 ext. 1D1 for fastest service Please allow 3 weeks for delivery

Quarterdeck

Quarterdeck Office Systems, 150 Pico Boulevard, Santa Monica, CA 90405 (310) 392-9851 Fax (310) 314-4219
Quarterdeck International Ltd., B.I.M. House, Crofton Terrace, Dun Laoghaire Co. Dublin, Ireland Tel.(353) (1) 284-1444 Fax: (353) (1) 284-4380

©1993 Quarterdeck Office Systems. Trademarks are property of their respective owners.

CIRCLE NO. 501 ON READER SERVICE CARD

Digital Video File Formats

Understanding QuickTime and Video for Windows

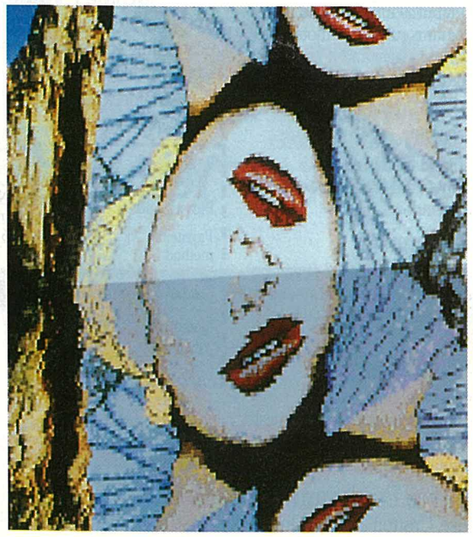
Mark Florence

Fads come and go, and like object-oriented programming, artificial intelligence, and bell-bottom dungarees, digital video is currently in vogue. Desktop digital video was pioneered — and proven — by Apple on the Macintosh. Over two years ago, QuickTime emerged as a strong standard with a loyal and talented following of developers. In late 1992, Apple announced QuickTime for Windows at the same time Microsoft ushered in Video for Windows, each vying to become desktop standards.

As of yet, there's no clear winner. But with the advent of powerful programs such as Adobe's Premiere for Windows, it's clear that digital video is approaching some stage of maturity. But what is digital video, and how can you, as a programmer, harness its power?

The 30,000-foot Perspective

Digital-video movies on your PC can be viewed as nothing more than a collection of rather large files that otherwise look like regular DOS files. Instead of holding eye-glazing columns of accounts-receivable data, however, these files contain digitized sequences of video and sound. (Incidentally, although I'll talk a lot about digital video, strictly speaking I'm referring to *time-based* data. For example, a QuickTime movie of a performance of *Tosca* might contain a track of video, a track of stereo sound, and additional text tracks of the libretto in English, German, and Italian, each synchronized to the other. Another point worth noting is that while the technocrats are well aware of the symmetry of using the Latin terms *video* ["I see"] and *audio* ["I hear"], *sound* is, for some reason, often preferred.)



An implementation of digital video must solve three problems to be viable. First, just like the analog systems that preceded it (CD-DA or NTSC for TV), it must define an architecture. This architecture must be robust enough to endure (so that content providers can be sure that their material won't quickly become obsolete) but flexible enough to adapt to the future. QuickTime and Video for Windows mainly embody their architectures in the data structures of their respective file formats (in the case of Video for Windows, this is called AVI, for audio-visual interleave). I'll examine these file formats in this article.

Secondly, digital video must provide extremely efficient compression and decompression implementations. A quick exercise in arithmetic shows why. Full-screen (say, 640×480), full-motion (either 24 frames per second in the movies, or 29.97 fps on your TV), uncompressed video needs between 22.1 and 27.6 Mbytes per second. One of my favorite movies, *The Great Escape*, would need 307 gigabytes to store uncompressed, which, if laid end-to-end...well, you get the picture. And space requirements are only part of the story. Imagine if mass-storage devices were cheap enough that you could afford 307 gigabytes for a single movie. Your hardware would still have to support a sustained data rate of 1 Mbyte per second to play it back.

Finally, a successful digital-video implementation must provide an engine that can play back the movies it digitizes at realistic frame sizes and rates on general-purpose desktop PCs. Both QuickTime and Video for Windows more or less succeed in this goal.

Mark is president of the San Francisco Canyon Company, which developed QuickTime for Windows for Apple Computer. Canyon publishes the Movie Toolkit, a C++ class library for manipulating QuickTime and AVI movie files, and Canyon Clipz!. Canyon's How to Digitize Video *will be published by John Wiley in early 1994. Mark can be reached at 415-398-9957; through CompuServe at 72371,104; or through AppleLink at CANYON.*

QuickTime Movie File Format

The first important point to note about the QuickTime movie file format is that QuickTime uses a strict subset of the Mac file format on the PC, making life easier for content providers. One consequence of this is that the byte ordering of structured data is Motorola, not Intel (because the Mac implementation came first). This may, at first, make it confusing to relate some of the discussion in this article to a hex dump of a QuickTime file (although I find Motorola hex easier to read!). In general, I'll still talk mostly about the PC, because it uses a simpler subset of the full QuickTime specification.

QuickTime files have a recursive, atom-based format. An atom is prefixed by a 32-bit length and a 32-bit identifier. It can contain either data or more atoms. Figure 1(a) shows the basic atom layout. The semantics of an atom are implied by its identifier. Each atom identifier is a four-character mnemonic, which is also the value of the identifier itself. This may seem odd to Windows programmers, to whom constructs like: *#define SOME_ATOM_ID 0x12345678 /* unreadable value */* are more familiar. On almost every other platform, 32-bit compilers have been quite happy to accept character constants like *moov* or *mdat*.

Clearly, a QuickTime movie can be viewed as a tree structure. Normally, of course, trees have a single root, but QuickTime movies have two. On the Mac, movie data (video frames and sound samples) is stored in the file's data fork; the atoms that describe that data can be stored in the resource fork. DOS, of course, does not have this concept, so both tree structures are concatenated, or *flattened* in QuickTime jargon. QuickTime on the Mac is quite happy to play these flattened movies.

Movie data is stored in the *mdat* atom, which always comes first. It contains only data, not other atoms, that data being the video frames and sound samples that comprise the actual movie. The *moov* atom (pronounced, *moo-vee*) is the root to a structure of atoms that act as an index to the movie data. Figure 1(b) shows the basic structure of all QuickTime movies on the PC.

The *moov* Atom

I mentioned earlier that atoms can be viewed as a tree. Table 1 shows the basic tree structure of the *moov* atom. While the semantics of a particular atom constrain it to a certain level in the tree, the ordering of atoms at a given level is arbitrary. Moreover, software that parses the tree is expected to ignore atoms it doesn't recognize. It is this simple facility that gives the QuickTime movie file structure the flexibility to adapt to future needs. You can explore this structure for yourself by using Apple's DUMPMOOV program. Under DOS, it generates output like that shown in Listing One, page 17. Space constraints prevent a detailed examination of each atom. This can be found in Apple's QuickTime Movie Exchange Toolkit, and Canyon's Movie Tookit.

The *mvhd* atom defines the overall characteristics of the movie, principally its time scale and duration. A *time scale* is simply the units (in events per second) in which time values are expressed. For example, a time scale of 1000 means that time values are interpreted as milliseconds. However, time scales of 100 or 1000, while seemingly convenient, are not often used. You will more likely see scales of 600 because it has more factors, allowing integer arithmetic to be performed with less loss of precision.

The Movie Header time scale and duration provide the key to synchronization. Rather than synchronize video and sound to a particular fixed frame rate (like analog systems or Microsoft's AVI), QuickTime synchronizes all its tracks to the Movie Header. In digital video (as opposed to analog video), frame rates do not have to be constant. There's no celluloid driven by sprockets in front of a beam of light. A single digital image can be displayed on the CRT for as short or as long a time as necessary. This is where the *stts* atom comes into play. Conceptually, it is an array of durations for each frame in the movie, each of which can, of course, be a different value. In practice, a simple compression scheme allows a single value to be applied to multiple frames. For example, Figure 2 specifies that all 1270 frames have a duration of 100.

QuickTime movies may contain an arbitrary number of *trak* atoms, reflecting each of its tracks. Like their analog counterparts, tracks can hold video or sound. Additionally, QuickTime supports text tracks, although they are not

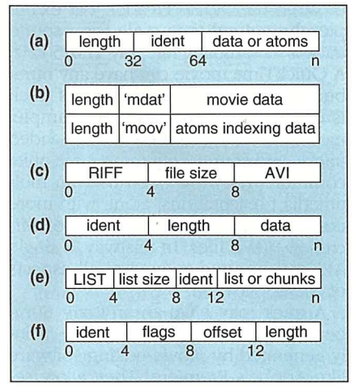


Figure 1: *(a) Basic atom format; (b) basic QuickTime movie file structure; (c) AVI RIFF form header; (d) basic chunk format; (e) list-header format; (f) AVI index-entry format.*

Atom	Purpose
moov	Movie atom.
-mvhd	Movie header. Defines the time scale and duration of the movie.
-trak	Track atom.
--tkhd	Track header. Defines the dimension, time scale, and duration of the track.
--edts	Edit list.
---elst	Edit-list entry. Allows selections of the track be played out of sequence.
--mdia	Media atom.
---mdhd	Media header. Defines the characteristics of the media holding this track's data.
---hdlr	Handler. On the Mac, defines the component that handles the media.
---minf	Media information.
----vmhd or smhd	Video- or sound-media information header. Defines basic media requirements.
----hdlr	Handler. On the Mac, defines the component that handles the video or sound.
----dinf	Data information.
-----dref	Data reference. On the Mac, can point to another file holding this track's data.
----stbl	Sample table.
-----stsd	Sample description. Describes how the track's video or sound is compressed.
-----stts	Time-to-sample. Gives the duration of each video frame.
-----stss	Sync sample. Indicates the location of key frames.
-----stsc	Sample-to-chunk. Groups video frames or sound samples into chunks.
-----stco	Chunk offset. Gives the offset into the *mdat* atom of each chunk.
-----stsz	Sample size. Gives the size of each video frame or sound sample.
-trak	As many additional tracks as required.

Table 1: moov *atom tree structure.*

yet implemented in the Windows version. Any number of video tracks can be present. QuickTime will choose the one that will look best when played on the target device. For example, you could digitize video into three tracks using 8-bit color, 16-bit color, or 24-bit (so-called) true color. If the movie is played on a PC with a video adapter capable of only 8-bit color, the 8-bit color track will be chosen. Similarly, any number of sound tracks may be present. Each can be recorded in a different language, for example. QuickTime will select the track that matches the current Windows language specification.

Tracks can, and typically do, have time scales and durations different from those in the Movie Header. For example, the natural time scale for a sound track is its sampling rate, say 11.025 kHz. A QuickTime movie can have any number of tracks, and no one type of track is conceptually favored. For example, movies are not required to have video tracks, and sound-only movies are quite common. They are often used in multimedia presentations, along with more conventional movies, instead of Microsoft WAVE files. In this way, a single API can be used to control all aspects of the application.

A track may have an arbitrary number of *elst* atoms. These atoms are mainly generated by movie-editing software like Adobe's Premiere. They allow selected parts of a track to be played out of sequence. You may have seen the recent Woody Allen movie, *Manhattan Murder Mystery*, in which Woody Allen and Diane Keaton attempt to blackmail their neighbor, whom they suspect of murder, by recording his girlfriend's audition for a play they've faked. Then they literally cut-and-paste the tape to produce a convincing, but quite different, shake-down message. In QuickTime, *elst* atoms are the digital equivalent of Woody's razor blade.

The *stsd* atom tells QuickTime how the track's video or sound data is compressed. The accompanying text box entitled, "Video Compression Technologies" explores this subject further. An additional text box entitled, "Selecting a Decompressor" describes how a decompressor is selected on playback.

A track may have multiple *stsd* atoms. At first, it's hard to see why this is useful. It implies that different parts of the track can be encoded using different compressors, and appears to be a somewhat esoteric feature. But consider a movie-editing package that might glue together parts of existing movies to form a new movie. If the source movies used different compressors, but multiple *stsd* atoms weren't supported, the new movie would have to be recompressed using a single compressor. However, each time a frame is compressed with a lossy compressor, it loses quality, much like a video tape that is copied.

Microsoft reportedly changed its mind during the Video for Windows beta period, and some early encodings record chunk offsets relative to the beginning of the file

The *stsc*, *stco*, and *stsz* atoms are used to extract data from the *mdat* atom. *stsc* allows video frames or sound samples to be grouped into chunks, to improve performance on playback. Typically, chunking is performed by postproduction optimization software. It gives the size of each chunk and the number of video frames or sound samples in that chunk. *stco* gives the offset of each chunk within the *mdat* atom. *stsz* gives the size of each video frame or sound sample.

The *mdat* Atom

The *mdat* atom is simply a stream of video frames and sound samples. Theoretically, the physical ordering is unimportant because the *stsc*, *stco*, and *stsz* atoms are used as indexes. However, in order to play back from relatively slow devices like CD-ROM, seeks must be avoided at all costs, so in practice, physical ordering is extremely important. QuickTime prefers that video frames and sound samples be physically grouped into half-second chunks, with sound leading. The text box entitled, "Sound Encoding Techniques" describes how sound samples are stored.

Reading and Writing QuickTime Movie Files

The first routine we'll need to tackle the task of reading and writing QuickTime movies is a fast WORD and DWORD flip routine, which converts Intel ordering to Motorola, and vice versa. Listing Two, page 18 shows *Flip16* and *Flip32*, both of which can be conveniently called from C/C++ code using the prototypes in Figure 3. In production code, you'll want to implement *Flip16Many* and *Flip32Many* in assembler so that you don't have to iterate over *Flip16* or *Flip32* to flip multiple WORDs or DWORDs.

I'll use recursive descent to parse QuickTime movies. This technique has the advantage of simplicity and elegance, as the structure of the code exactly matches the structure of the file itself. Listing Three, page 18, shows the *CollectAtomsFromFile* routine, the heart of the recursive-descent logic. Listing Four, page 18, shows the actual parsing code. I use the Windows multimedia I/O calls (*mmioOpen*, *mmioSeek*, *mmioRead*, and *mmioClose*) for convenience; in this context, they're equivalent to any other I/O interface. I use the Windows *mmioFOURCC* macro to construct atom identifier constants; if you had a 32-bit compiler (as for Windows NT), you could simply code *moov*, for example, directly.

You may have noticed that *CollectAtomsFromFile* flips the atom size *atmb.lSize* (in order to perform arithmetic) but not its identifier *atmb.lName*. This isn't a bug. Rather, it allows you to code *mmioFOURCC* constants in their natural, readable Motorola order.

Although there isn't room here to show the actual code, writing a QuickTime movie uses a structurally parallel technique. I use the following procedure:

1. Write out a dummy *mdat* atom with a zero length.

```
stts (24) Time To Sample
-Version/Flags: 0x00000000
-Number Of Entries: 1
- 0: Sample Count 1270, Sample Duration 100.
```

Figure 2: *Applying a duration value to multiple frames.*

```
WORD PASCAL Flip16 (WORD);
DWORD PASCAL Flip32 (DWORD);
```

Figure 3: *Prototypes for the* Flip16 *and* Flip32 *routines, which converts Intel ordering to and from Motorola.*

2. Write out all the movie data (video frames and sound chunks) in the desired order.
3. At the same time, accumulate, in internal tables, the information you'll need to build the *moov* atoms.
4. Seek to the beginning of the file, and then write out the true length of the *mdat* atom.
5. Seek to the end of the *mdat* atom.
6. Write out all the *moov* atoms from your internal tables. Mirror the recursive-descent technique to write leaf atoms first, working up the tree toward the root. Each routine that writes an atom returns its length. This way, routines that write nonleaf atoms simply accumulate their length from the routines they call.

AVI Movie File Format

AVI files are stored as a specialization (*form* in Microsoft jargon) of the Microsoft RIFF (Resource Interchange File Format) standard. Microsoft defines RIFF as a tagged-file specification used to define standard formats for multimedia files. Other forms are WAVE for waveform audio data and RDIB for bitmaps. An introduction to RIFF can be found in the *Windows Multimedia Programmer's Guide*, and a discussion of the AVI form can be found in the *Video for Windows Development Kit Programmer's Guide*.

The AVI RIFF form starts with a standard 12-byte header; see Figure 1(c). Of course, Intel byte ordering is used for all fields. In order to code identifiers, such as RIFF and AVI, naturally, Microsoft provides the *mmioFOURCC* macro. For example, the following type of construct is common: *#define formtypeAVI mmioFOURCC('A', 'V', 'I', ' ').*

In general, RIFF files consist of *chunks*, lists of chunks, or a combination of both. The AVI form specifies which chunks are defined and the order in which they are expected. All programs that read RIFF files are expected to ignore chunks they don't recognize (but preserve them when the file is written). A chunk is very similar in both form and concept to a QuickTime atom; it consists of a 4-byte identifier and a 4-byte length, followed by the chunk data; see Figure 1(d). The semantics of a chunk or list are implied by its identifier. A list of chunks is prefixed by a 12-byte header, as in Figure 1(e).

Microsoft supplies two good programs for exploring AVI files. RIFFWALK works under DOS, and generates output like that shown in Listing Five, page 19. It's worth taking a look at this code; armed with the information discussed so far, you'll be able to infer a lot about the AVI file structure. FILEWALK displays similar output under Windows. Table 2 shows the required chunks and lists of chunks in an AVI file. I'll discuss the highlights of the important chunks shortly. Unlike QuickTime atoms, the ordering of AVI chunks is important.

The *hdrl* List

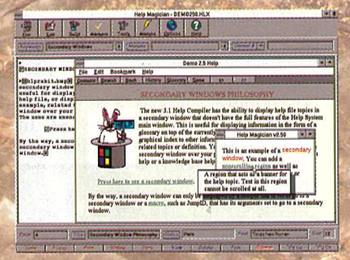
The *hdrl* list must come first in the AVI file. Its function is analogous to that of the QuickTime *moov* atom. The *avih* chunk defines the overall characteristics of the movie, principally the number of *streams* (more conventionally known as "tracks") the movie contains, and the frame rate and size of the video. This scheme has three problems. First, it is biased towards video. It is conceptually impossible for an AVI file to *not* have a video track (like a QuickTime sound-

Code			Purpose
RIFF	AVI		File header.
	LIST	hdrl	Defines structure of data in the *movi* list.
		avih	Defines basic movie format.
		LIST strl	One *strl* list per stream (video or sound data)
		strh	Defines stream format.
		strf	
		LIST strl	
		strh	
		strf	
		...	
	junk		Optionally, provides padding (otherwise ignored).
		LIST movi	Contains actual movie data.
		LIST rec	Groups video and sound data for efficiency.
		##wb	Sound data.
		##dc	Video data.
		LIST rec	
		##wb	
		##dc	
		...	
	idx1		An optional index into *movi* list.

Table 2: *AVI file structure.*

Windows Help Magician 2.5

PC MAGAZINE says "Designing help systems with the Help Magician is so easy that you might even want to use it as a hypertext authoring tool, independent of providing help for a product. Certainly for Windows programmers it's indispensable"! - July 1993 Toolkits section



Create, edit, and test help quickly in an integrated environment!

We've sliced the hypertext help development time from days to hours! Forget about footnote or RTF programming or using Word. The Help Magician comes with a 30-day money back guarantee.

New version 2.5 features

- ❑ No memory limitations
- ❑ Supports 3.1 Help Compiler features such as non-scrolling regions and secondary windows
- ❑ Set background colors
- ❑ Advanced paragraph formatting
- ❑ Full macro support
- ❑ Jump to other help files
- ❑ Multiple file support
- ❑ Help window sizing
- ❑ Imports Word & Lotus Ami Pro RTF files
- ❑ Built-in spell checker
- ❑ Multimedia support for audio, video, midi, and animation files
- ❑ Writes #define files
- ❑ Still only $199!

Order Now! Call (800) 542-2742

SOFTWARE INTERPHASE INCORPORATED

82 Cucumber Hill Rd
Suite 111
Foster, RI 02825
Voice: (401) 397-2340
Fax: (401) 397-6814

CIRCLE NO. 500 ON READER SERVICE CARD

only movie). Second, all video tracks must have frames of identical size. Third, AVI movies are bound to the old analog concept of a fixed frame rate. As we noted earlier, digital-video engines are free to display frames for as long or as short a time as they like.

Recall that CD-ROM transfer speed is currently a tremendous limiting factor on playback rate, regardless of the rate at which the movie was recorded and the speed of the decompressor. For example, a common CD-ROM drive can sustain a transfer rate of about 150 Kbytes per second. Simple arithmetic shows that if each frame is 10 Kbytes, a maximum 15 fps frame rate can be expected. Consequently, some sophisticated postproduction software has been written to limit the data rate of digital-video movies. Apple's MovieShop for QuickTime does a decent job all the way down to 90 Kbytes per second (for single-speed CD-ROM playback). At the risk of oversimplifying, one technique it may use is to combine similar frames. To preserve sound synchronization it simply adjusts the length of time the frame is displayed in the *stts* atom. This technique cannot be used for AVI movies.

The *hdrl* list contains one *strl* list for each track. Currently, Video for Windows requires exactly one video track and at most one sound track. The ordering of the *hdrl* lists is unimportant; however (for reasons that will become apparent when we look at the *movi* list), the first is denoted as stream 00, the second as stream 01, and so on. The *strh* and *strf* further define the characteristics of each track (such as the sampling rate and size for sound).

Video and sound tracks are not synchronized to a common time base. Instead, video is expected to be played at its frame rate, and sound at its sampling rate, with some element of faith that they will match up. This simple scheme does not allow, for example, effects such as discontinuous sound without artificially inserting periods of silence (and increasing the data rate).

The *movi* List

The *movi* list follows the *hdrl* list, and contains the actual movie data. It is analogous to the QuickTime *mdat* atom. Often, the *movi* list is preceded by a so-called "junk" chunk so that the first data chunk is aligned on a 2K boundary, improving playback performance from CD-ROM. These chunks are used only for alignment and have no other semantics.

Video Compression Technologies

Table 3 summarizes the compression technologies available today for QuickTime and Video for Windows. By the time you read this, more may be known about the Captain Crunch and Indeo R3 compressors (both still in beta at the time of this writing). Those compressors producing encodings that are identical in both systems are indicated with an asterisk. I have deliberately omitted MPEG, motion, JPEG, and other technologies, simply because no QuickTime or Video for Windows CODECs exist yet.

I've also indicated a typical frame size and rate, although these numbers should be taken with a grain of salt. I've assumed software-only decompression on a 486/33 PC. In my opinion, the current leader of the pack is clearly CinePak, although Captain Crunch and Indeo R3 show signs of catching up.

When analyzing the performance of a CODEC, the most important gating factor is the CD-ROM transfer rate, because most movies are distributed this way. Consider that common CD-ROMs have 150–200 Kbytes/second transfer rates. A good CODEC will attempt to compress the data as tightly as possible (which gates the *maximum* playback rate from CD) in such a way that it can be decoded as quickly as possible (which gates the actual playback rate). For example, if the average frame size is 10K, then no more than 15 to 20 fps from CD can be expected, regardless of the speed of the decompressor.

—M.F.

Compressor	Identifier	Frame Size/Rate	Comments
Apple Animation	rle	320x240/12 fps	Optimized for animations and cartoons. Gives poor performance for real-life video content.
Apple Graphics	smc	160x120/15 fps	A modest performer optimized for 8-bit content. (The identifier is the initials of its patent holder, Sean Callaghan.)
Apple Video	rpza	160x120/15 fps	Also known as "road pizza" because of its good compression ratio, it is now superceded by CinePak.
Captain Crunch*	klic	320x240/15 fps	From MediaVision; currently in beta.
CinePak*	cvid	320x240/15 fps	The one apparent disadvantage currently is that the algorithm is highly asymetrical. It takes up to 100 seconds to compress one second of video. For content with a high turnover and a short life, this can be critical.
Intel Indeo R2*	rt21	160x120/12 fps	A modest performer without hardware assistance. When available, will be superceded by Indeo R3. Intel's ISVR card captures directly into this format.
Intel Indeo R3*	iv31	320x240/15 fps	Currently in beta.
Intel YVU9	yvu9		Primarily used only during capture; available only in Video for Windows. Content is almost always converted into another format.
JPEG	jpeg		Primarily used only during capture; available only in QuickTime. Content almost always converted into another format.
Microsoft RLE	mrle	160x120/12 fps	Optimized for animations and cartoons. Gives poor performance for real-life video content. Not the same as Apple's Animation compressor.
Microsoft Video 1*	msvc	160x120/12 fps	Media Vision's ProMovie Studio captures directly into this format.

Table 3: *Compression technologies currently available for QuickTime and Video for Windows (*common to both QuickTime for Windows and AVI).*

Data in the *movi* list can be structured either in chunks or in lists of chunks. Sound data is stored in *##wb* chunks, while video data is stored in *##dc* chunks, where *##* represents the corresponding stream number. Video for Windows prefers that sound and video chunks be paired, with the video chunk holding a frame and the corresponding sound chunk holding a frame's worth of sound.

For efficiency, these sound-video chunk pairs can themselves be grouped inside a *rec* list. The playback engine will read the entire contents of a *rec* list at once. Often, the list will end with a junk chunk so that its length is a multiple of 2K.

The *idx1* Chunk

Although it is technically optional, almost all AVI movies end with an index (*idx1*) chunk. Each entry in the index points to a chunk or rec list in the *movi* list. Figure 1(f) shows the AVI index-entry format. If the index is present (as denoted by flags in the *avih* chunk), you are expected to use it to parse the data in the *movi* list. The ordering of index entries defines the order in which video and sound chunks must be played. One trick about using the index is worth noting. It normally records chunk offsets relative to the start of the *movi* list. However, Microsoft reportedly changed its mind during the Video for Windows beta period and some early encodings record chunk offsets relative to the beginning of the file. To determine which is being

Selecting a Decompressor

Key to the flexibility of both QuickTime and Video for Windows is their open architecture for compressors and decompressors (CODECs). Today, vector-quantization compression techniques allow playback rates of approximately 12 to 15 fps of 240×180 frames on most general-purpose computers. Tomorrow, perhaps wavelets or fractals will double this. It's vital that both QuickTime and Video for Windows accommodate this growth without changing their file formats or architecture. Fortunately, they do, and we are starting to see a wide range of powerful CODECs from Apple, Microsoft, and third-party developers.

QuickTime decompressors are structured as *components*. Components are a Mac concept, ported to the PC in QuickTime for Windows. A component is a special kind of DLL (in Windows, they normally use the .QTC extension) that negotiates its capabilities with its callers through a predefined set of entry points. A single .QTC file, which Windows views as a DLL, can contain multiple components. Full details are in Apple's QuickTime documentation.

You may recall from the general discussion that the *stsd* atom describes how a track's video is compressed. It does this by encoding the four-character identifier of the compressor. The assignment of these identifiers is regulated by Apple to ensure that they remain unique across all third-party developers. They look just like atom identifiers; for example, *cvid* is assigned to SuperMac's CinePak CODEC.

When QuickTime for Windows starts to play a video track, it negotiates with all the decompressor components it can find, using the standard Windows *LoadLibrary* search strategy. Each decompressor is asked, of course, if it can handle the identified encoding. But it also has the opportunity to check if a preferred environment (for example, special hardware) is present. In any event, it will report whether or not it can perform the decompression and, if so, how fast. The speed being measured as the number of milliseconds necessary to decode a 320×240 frame. QuickTime then uses the fastest decompressor.

Even when a movie is playing, QuickTime can switch decompressors. For example, if the video frame becomes clipped by another window, QuickTime will repeat the decompressor selection process. It does this because a decompressor that uses hardware assistance may wish to defer to a software-only decompressor for nonrectangular frames.

This elegant scheme is simple and effective, although it does place a burden on the decompressor writer to develop the correct negotiation logic. It has the advantage in that decompressors can simply be dropped into the user's system without any SYSTEM.INI changes. For example, content providers can deliver a CD of movies and a proprietary QuickTime decompressor without fear of a conflict with the existing environment or special installation requirements. Further, multiple decompressors for the same encoding can be present, and QuickTime will automatically choose the most appropriate.

```
[drivers]
VIDC.MSVC=msvidc.drv
VIDC.YVU9=isvy.drv
VIDC.IV31=indeor3.drv
VIDC.RT21=indeo.drv
VIDC.CVID=iccvid.drv
VIDC.MRLE=msrle.drv
```

Figure 4: *Example [drivers] section of a Windows SYSTEM.INI file.*

Video for Windows decompressors are drivers (DLLs with the extension .DRV) written to the specification Microsoft documents in the *Video for Windows Development Kit Programmer's Guide*. In a manner similar to QuickTime, the AVI file format encodes the four-character identifier of the compressor in the video stream header, *strh*. Again, the assignment of these identifiers is regulated by Microsoft to ensure uniqueness, although we can be sure that the level of coordination between Apple and Microsoft is fairly low! Fortunately, where an encoding is supported in both systems, its identifier is constant. For example, Microsoft has also assigned *cvid* to SuperMac's CinePak CODEC.

Before it plays a video stream, Video for Windows simply takes the encoder identifier, prefixes it with VIDC, and uses it to look up the name of the CODEC in the *[drivers]* section of SYSTEM.INI; see the *[drivers]* section in Figure 4.

The scheme is simple and effective, but it has disadvantages compared to QuickTime. An installation procedure of some kind is required, and multiple decompressors for the same encoding cannot coexist. This means that a new version of a decompressor cannot specialize the capabilities of existing versions; it must totally replace them. Imagine that Intel wants to develop a new version of the Indeo decompressor optimized especially for the XYZ video chipset. Under QuickTime, it need only perform this one task, and can defer to other decompressors if the XYZ chip is not present. Video for Windows decompressors under Video for Windows must assume all the functionality of prior versions.

—M.F.

Sound Encoding Techniques

Both QuickTime and AVI formats store sound in similar ways. At the time of this writing, neither supported compressed sound. Table 4 summarizes the encoding techniques each uses.

When sound is digitized, analog signals are converted to numbers. The size of those numbers is referred to as the *sample size*. The rate at which the analog signal is sampled to is called the *sample rate*. In general, the larger the sample size and rate, the better the quality of the digitization. As a point of reference, CD-DA (standard audio CDs) is the equivalent of 16-bit, 44.1 kHz sound.

For sample sizes of 8 bits, each sample represents one of possibly 256 different values; for 16-bit samples, 65,536 discrete values can be represented. You might visualize the difference in quality to be analogous to that more easily perceived between 8- and 16-bit color. Each sample represents the deviation of a waveform from a midpoint. Two conventions exist for the midpoint. In AVI, 8-bit samples use 0x80 as the midpoint (the so-called "raw format"), and 16-bit samples use conventional signed numbers with 0x0000 as the midpoint (so-called "two's-complement" format). QuickTime can use either format with either sample size Figure 5 shows this more clearly.

To complicate matters a little, Microsoft does not actually use the jargon raw and two's complement. Instead, it uses the acronym PCM to refer to its 8-bit and 16-bit encodings. To convert between the two formats, simply XOR each sample with 0x80 or 0x8000 as appropriate.

QuickTime stores 16-bit sound samples in Motorola order; AVI uses Intel order. Byte ordering is, of course, moot for 8-bit samples! Consequently, 16-bit sound samples must be flipped when converting from AVI to QuickTime and vice versa.

Most PC sound cards can only digitize and play back at the three standard MPC rates of 11.025, 22.05, and 44.1 kHz. Many QuickTime movies are captured on the Mac and their sample rates can appear as weird numbers like 11.12754 kHz. Both QuickTime and AVI share the same convention for stereo sound in that the left-channel sample appears before the right-channel sample in the stream.

The interplay of sample size, rate, and number of channels has a great effect on the ability of the QuickTime or AVI engine to playback a movie. For example, CD-DA quality sound (16-bit, 44.1 kHz, stereo) requires a sustained data transfer rate of 176.4 Kbytes per second. Single-speed CD-ROM drives are capable of a peak rate of 150 Kbytes per second, which doesn't leave a lot of room for video! For this reason, most digital video movies you'll see today use 8-bit, 11-kHz mono sound (which doesn't sound too bad through most PC speakers). This situation is unlikely to improve much until we see a quantum leap in hardware performance.

Interleave is a primary characteristic of digital video files, so much so that the AVI file format is named after the concept. However, interleave is mainly a factor for slow playback devices such as CD-ROM. The trick that both the QuickTime and AVI engines have had to master is to stream enough data from the CD-ROM to keep themselves busy. It is a delicate balance of RAM buffer sizes, transfer rates, seek times, and playback rate. Note that *streaming* is actually quite the opposite of the more conventional *caching*. A cache (like SMARTDrive) attempts to improve performance by anticipating that data, once read, will be read again. Streaming assumes that data will be read once, from beginning to end, and attempts to steadily supply that data at the same rate that it is consumed.

Although QuickTime and AVI acknowledge the same concept, their engines have different requirements for interleave. QuickTime prefers sound and video in half-second chunks, with sound leading. AVI prefers sound and video interleaved on a frame-by-frame basis. That is, each video frame is physically followed by a frames worth of sound. To complicate matters, though, sound samples are skewed ahead of video by 0.75 second. In a dump of an AVI file, you'll see the first few sound samples unmatched by video frames, and the last few video frames unmatched by sound samples (look at the end of Listing Five (page 19) for an example of this).

When an AVI file is converted to QuickTime, or vice versa, the interleave factor should be adjusted to these preferred values. If it is not, you can expect poor playback performance from a CD-ROM.

—M.F.

	QuickTime	AVI
Sample Size	8 bit, 16 bit.	8 bit, 16 bit.
Sample Rate	Continuum of rates, normally between 11.0 and 44.1 kHz.	Normally the discrete MPC rates of 11.025, 22.05, and 44.1 kHz.
Channels	Mono and stereo.	Mono and stereo.
Interleave	Half-second chunks, sound leading.	Frame-by-frame, sound skewed. ahead by 0.75 second.

Table 4: *Sound-encoding techniques.*

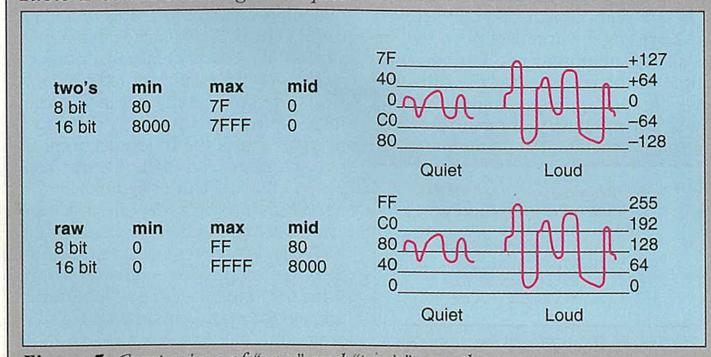


Figure 5: *Comparison of "raw" and "two's" sound.*

SEND NO MONEY
SUBSCRIBE TODAY

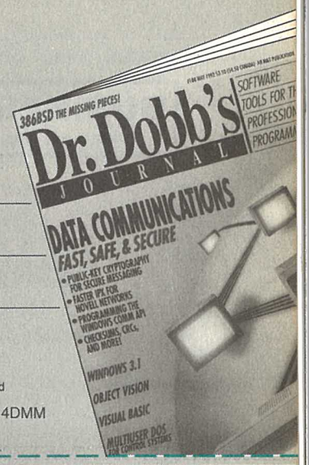
"PROGRAMMING POINTERS UNVEILED"

YES! Enter my one-year subscription to ***Dr. Dobb's Journal*** (12 issues) for only $25. I save 47% off the cover price!

NAME
ADDRESS
CITY/STATE/ZIP

☐ Payment Enclosed ☐ Bill Me Later

Savings based on the full one-year newsstand rate of $47.40. Foreign subscriptions must be prepaid in U.S. dollars drawn on a U.S. bank. Canada and Mexico: $45/year (surface mail). All other foreign countries: $70/year (air mail). California residents please add 8.25% sales tax. Canadian GST included (#R124771239). Please allow 4-6 weeks for delivery of first issue. 4DMM



"THE LATEST ON EMERGING TECHNOLOGIES"

YES! Enter my one-year subscription to ***Dr. Dobb's Journal*** (12 issues) for only $25. I save 47% off the cover price!

NAME
ADDRESS
CITY/STATE/ZIP

☐ Payment Enclosed ☐ Bill Me Later

Savings based on the full one-year newsstand rate of $47.40. Foreign subscriptions must be prepaid in U.S. dollars drawn on a U.S. bank. Canada and Mexico: $45/year (surface mail). All other foreign countries: $70/year (air mail). California residents please add 8.25% sales tax. Canadian GST included (#R124771239). Please allow 4-6 weeks for delivery of first issue. 4DMM

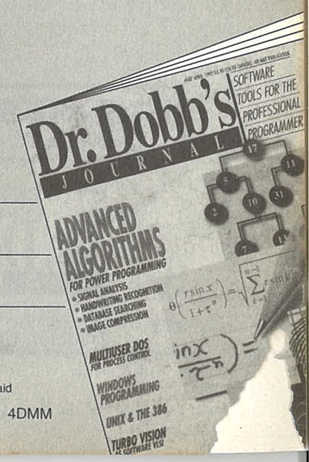
"TOMORROW'S SOFTWARE TOOLS TODAY"

YES! Enter my one-year subscription to ***Dr. Dobb's Journal*** (12 issues) for only $25. I save 47% off the cover price!

NAME
ADDRESS
CITY/STATE/ZIP

☐ Payment Enclosed ☐ Bill Me Later

Savings based on the full one-year newsstand rate of $47.40. Foreign subscriptions must be prepaid in U.S. dollars drawn on a U.S. bank. Canada and Mexico: $45/year (surface mail). All other foreign countries: $70/year (air mail). California residents please add 8.25% sales tax. Canadian GST included (#R124771239). Please allow 4-6 weeks for delivery of first issue. 4DMM



Recycled Paper

PROCESS IMMEDIATELY

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 1194 BOULDER, CO

POSTAGE WILL BE PAID BY ADDRESSEE

Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

P.O. Box 52226
Boulder, CO 80321-2226

PROCESS IMMEDIATELY

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 1194 BOULDER, CO

POSTAGE WILL BE PAID BY ADDRESSEE

Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

P.O. Box 52226
Boulder, CO 80321-2226

PROCESS IMMEDIATELY

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 1194 BOULDER, CO

POSTAGE WILL BE PAID BY ADDRESSEE

Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

P.O. Box 52226
Boulder, CO 80321-2226

SUBSCRIBE TODAY
SEND NO MONEY

used, I read the first index entry. If its chunk offset is large (greater than 2K), I assume the old encoding; if it is small, I assume the new.

Reading and Writing AVI Movie Files

The Windows multimedia I/O calls (*mmioOpen*, *mmioClose*, *mmioRead*, *mmioWrite*, *mmioSeek*, *mmioDescend*, and *mmioAscend*) are designed to process RIFF files. In particular, *mmioDescend* and *mmioAscend* allow chunks and lists of chunks to be processed quite conveniently. As a point of comparison, QuickTime provides no such assistance, and dealing with AVI files is considerably simpler.

Listing Six (page 19) shows how to parse a basic AVI movie file. For clarity, error checking has been omitted. Again, there isn't room to show the code for writing an AVI file, but I use this technique.

1. Seek to an offset of 2K into the file.
2. Write out all the movie data (video frames and sound chunks) in the desired order.
3. At the same time, accumulate, in internal tables, the information you'll need to build the *hdrl* list.
4. Seek back to the beginning of the file and create a RIFF chunk and the required chunks in the *hdrl* list.
5. Create a junk chunk to pad the end of the *hdrl* list to the beginning of the *movi* list.
6. Create a *movi* list chunk.
7. Seek to the end of the file and create an index chunk.

Conclusion

Content developers often ask whether they should develop for QuickTime or Video for Windows. On the one hand, I think that QuickTime is technically superior. As far as production is concerned, the Intel Smart Video Recorder (ISVR card) can capture QuickTime and AVI movies with equal ease, and products like Adobe's Premiere bring first-rate editing capabilities to both. And on the Mac, where Video for Windows is not even a player, there exists a vast pool of equipment, software, and (most importantly) production talent, all dedicated to QuickTime.

On the other hand, Apple is fast losing ground to Microsoft by daring to play in Microsoft's sandbox. The decision to do Windows was a bold one, but unless Apple begins exhibiting a much stronger commitment to QuickTime for Windows, it may ultimately be overwhelmed.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

For More Information

QuickTime Developer's Kit
Apple Computer
P.O. Box 319
Buffalo, NY 14207
800-282-2732
$195

Video for Windows
Microsoft Corp.
One Microsoft Way
Redmond, WA 98052-6399
Available free on CompuServe

Canyon Movie Toolkit
San Francisco Canyon Company
150 Post Street, Suite 620
San Francisco, CA 94108
415-398-9957
$795

Listing One *(Text begins on page 10.)*

```
moov (16658) Movie Atom
  mvhd (108) Movie Header
  -Version/Flags: 0x00000000
  -Creation Time: Thu Aug 19 13:26:31 1993
  -Modification Time: Thu Aug 19 13:26:31 1993
  -Time Scale: 10000 per second
  -Duration: 127000
  -Preferred Rate: 1
  -Preferred Volume: 0x00ff
  -Matrix:       1          0          0
                 0          1          0
                 0          0          1
  -Preview Time: 0
  -Preview Duration: 0
  -Poster Time: 0
  -Selection Time: 0
  -Selection Duration: 0
  -Current Time: 0
  -Next Track ID: 2
  trak (5524) Track Atom
    tkhd (92) Track Header
    -Version/Flags: 0x0000000f
    -Creation Time: Thu Aug 19 13:26:31 1993
    -Modification Time: Thu Aug 19 13:26:31 1993
    -Track ID: 0
    -Time Scale: 0 per second
    -Duration: 127000
    -Movie Time Offset: 0
    -Priority: 0
    -Layer: 0
    -Alternate Group: 0
    -Volume: 0
    -Matrix:       1          0          0
                   0          1          0
                   0          0          1
    -Track Width: 320
    -Track Height: 240
    edts (36) Edit List
      elst (28) Edit Entry
      -Version/Flags: 0x00000000
      -Number Of Entries: 1
      - Entry 0: Duration 127000, time 0, rate 1.
    mdia (5388) Media Atom
      mdhd (32) Media Header
      -Version/Flags: 0x00000000
      -Creation Time: Thu Aug 19 13:26:31 1993
      -Modification Time: Thu Aug 19 13:26:31 1993
      -Time Scale: 11025 per second
      -Duration: 1400175
      -Language: 0x0000
      -Quality: 0x0000
      hdlr (32) Handler
      -Version/Flags: 0x00000000
      -Component Type: mhlr
      -Component Subtype: soun
      -Component Manufacturer: appl
      -Component Flags: 0x00000000
      -Component Flags Mask: 0x00000000
      minf (5316) Video Media Info
        smhd (16) Sound Media Information
        -Version/Flags: 0x00000000
        -Balance: 0
        hdlr (32) Handler
        -Version/Flags: 0x00000000
        -Component Type: dhlr
        -Component Subtype: alis
        -Component Manufacturer: appl
        -Component Flags: 0x00000000
        -Component Flags Mask: 0x00000000
        dinf (36) Data Info
          dref (28) 00 00 00 00 00 00 00 00 01 00 00 00 0c 61 6c 69 73
                    00 00 00 01
        stbl (5224) Sample Table
          stsd (52) Sample Description
          -Version/Flags: 0x00000000
          -Number Of Entries: 1
            raw  (36) Sound Description
            -Data reference ID: 0x0000
            -Version: 0x0000
            -Codec Revision Level: 0x0000
            -Codec Vendor: appl
            -Number of Channels: 1
            -Bits/Sample: 8
            -Compression ID: 0
            -Packet Size: 0
            -Sample Rate: 11025.
          stts (24) Time To Sample
          -Version/Flags: 0x00000000
          -Number Of Entries: 1
          - 0: Sample Count 1400175, Sample Duration 1.
          stsc (3832) Sample To Chunk
          -Version/Flags: 0x00000000
          -Number Of Entries: 318
          - 0: First Chunk 1, Sample per Chunk 4410, Chunk Tag 1.
          ...
          - 317: First Chunk 318, Sample per Chunk 1500, Chunk Tag 1.
          stsz (20) Sample Size
          -Version/Flags: 0x00000000
          -Sample Size: 1
          -Number Of Entries: 1400175
          stco (1288) Chunk Offset
          -Version/Flags: 0x00000000
          -Number Of Entries: 318
```

(continued on page 18)

Listing One *(Listing continued, text begins on page 10.)*

```
            8 18268 71942 125877 181638 239367 295397 351595 408044 466170
        -Dumping 1232 bytes
trak (11018) Track Atom
  tkhd (92) Track Header
   -Version/Flags: 0x0000000f
   -Creation Time: Thu Aug 19 13:26:31 1993
   -Modification Time: Thu Aug 19 13:26:31 1993
   -Track ID: 1
   -Time Scale: 1000 per second
   -Duration: 127000
   -Movie Time Offset: 0
   -Priority: 0
   -Layer: 0
   -Alternate Group: 0
   -Volume: 0
   -Matrix:          1          0          0
                     0          1          0
                     0          0          1
   -Track Width: 320
   -Track Height: 240
   edts (36) Edit List
     elst (28) Edit Entry
      -Version/Flags: 0x00000000
      -Number Of Entries: 1
      - Entry 0: Duration 127000, time 0, rate 1.
   mdia (10882) Media Atom
     mdhd (32) Media Header
      -Version/Flags: 0x00000000
      -Creation Time: Thu Aug 19 13:26:31 1993
      -Modification Time: Thu Aug 19 13:26:31 1993
      -Time Scale: 1000 per second
      -Duration: 127000
      -Language: 0x0000
      -Quality: 0x0000
     hdlr (32) Handler
      -Version/Flags: 0x00000000
      -Component Type: mhlr
      -Component Subtype: vide
      -Component Manufacturer: appl
      -Component Flags: 0x00000000
      -Component Flags Mask: 0x00000000
     minf (10810) Video Media Info
       vmhd (20) Video Media Information Header
        -Version/Flags: 0x00000000
        -Graphics Mode: 64
        -Op Color: 0x0000, 0x0000, 0x0000
       hdlr (32) Handler
        -Version/Flags: 0x00000000
        -Component Type: dhlr
        -Component Subtype: alis
        -Component Manufacturer: appl
        -Component Flags: 0x00000000
        -Component Flags Mask: 0x00000000
       dinf (36) Data Info
         dref (28) 00 00 00 00 00 00 00 01 00 00 00 0c 61 6c 69 73 00 00 00 01
       stbl (10714) Sample Table
         stsd (102) Sample Description
         -Version/Flags: 0x00000000
         -Number Of Entries: 1
           cvid (86) Image Description (cvid)
           -Version: 1
           -Revision Level: 1
           -Vendor: appl
           -Temporal Quality: 0x3ff
           -Spatial Quality: 0x3ff
           -Width (in pixels): 320
           -Height (in pixels): 240
           -Horizontal Resolution: 72
           -Vertical Resolution: 72
           -Data Size: 0
           -Codec name: Movie Toolkit (cvid)
           -Depth: 24
           -Dumping 2 bytes
         stts (24) Time To Sample
         -Version/Flags: 0x00000000
         -Number Of Entries: 1
         - 0: Sample Count 1270, Sample Duration 100.
         stss (356) Sync Sample
         -Version/Flags: 0x00000000
         -Number Of Entries: 85
              1 16 31 46 61 76 91 106 121 136
         -Dumping 300 bytes
         stsc (28) Sample To Chunk
         -Version/Flags: 0x00000000
         -Number Of Entries: 1
         - 0: First Chunk 1, Sample per Chunk 1, Chunk Tag 1.
         stsz (5100) Sample Size
         -Version/Flags: 0x00000000
         -Sample Size: 0
         -Number Of Entries: 1270
               13850 12357 12148 12439 12320 12338 12323 12481 12383 12499
         -Dumping 5040 bytes
         stco (5096) Chunk Offset
         -Version/Flags: 0x00000000
         -Number Of Entries: 1270
              4418 22678 35035 47183 59622 76352 88690 101013 113494 130287
         -Dumping 5040 bytes
```

Listing Two

```
        ALIGN 16
Flip16  PROC    FAR VALUE:WORD
        MOV     AX, VALUE
        ROL     AX, 8
        RET
Flip16  ENDP


        ALIGN   16
Flip32  PROC    FAR VALUE:DWORD
        MOV     DH, BYTE PTR VALUE
        MOV     DL, BYTE PTR VALUE + 1
        MOV     AH, BYTE PTR VALUE + 2
        MOV     AL, BYTE PTR VALUE + 3
        RET
Flip32  ENDP
```

Listing Three

```
typedef int (*ATOMFILPARSER) (HMMIO hmmio, long lName, long lOffset, long
lSize);

int CollectAtomsFromFile (HMMIO hmmio, long lOffset, long lSize,
                                                    ATOMFILPARSER apfil) {
    struct {long lSize, lName; } atmh;

    // Process the various atoms as we find them
    for (; lSize > 0; lOffset += atmh.lSize, lSize -= atmh.lSize)  {
        if (lSize < sizeof atmh)
            return FALSE ;
        mmioSeek (hmmio, lOffset, SEEK_SET);
        if (mmioRead (hmmio, (HPSTR) &atmh, sizeof atmh) != sizeof atmh)
            return FALSE ;
        atmh.lSize = Flip32 (atmh.lSize);
        if (atmh.lSize < sizeof atmh)
            return FALSE ;
        if (! apfil (hmmio, atmh.lName, lOffset+sizeof atmh,
                                                    atmh.lSize-sizeof atmh))
            return FALSE ;
    }
    // If the movie is well-formed, we should end on an atom boundary
    return (lSize == 0);
}
```

Listing Four

```
 ...
hmmio = mmioOpen (szFileName, NULL, MMIO_READ | MMIO_DENYNONE);
CollectAtomsFromFile (hmmio, 0, mmioSeek (hmmio, 0, SEEK_END),
&ParseWholeMovie);
mmioClose (hmmio);
 ...

int ParseWholeMovie (HMMIO hmmio, long lName, long lOffset, long lSize)
{
    switch (lName) {
        case mmioFOURCC ('m','o','o','v'):
            return CollectAtomsFromFile (hmmio, lOffset, lSize,
&ParseMoovAtom);
        default:
            return TRUE;
    }
}
int ParseMoovAtom (HMMIO hmmio, long lName, long lOffset, long lSize) {
    switch (lName) {
        case mmioFOURCC ('m','v','h','d'):
         /* your code */
            return TRUE;
        case mmioFOURCC ('t','r','a','k'):
            return CollectAtomsFromFile (hmmio, lOffset, lSize, &ParseTrakAtom);
        default:
            return TRUE;
    }
}
int ParseTrakAtom (HMMIO hmmio, long lName, long lOffset, long lSize) {
    switch (lName) {
        case mmioFOURCC ('t','k','h','d'):
         /* your code */
            return TRUE;
        case mmioFOURCC ('e','d','t','s'):
            return CollectAtomsFromFile (hmmio, lOffset, lSize, &ParseEdtsAtom);
        case mmioFOURCC ('m','d','i','a'):
            return CollectAtomsFromFile (hmmio, lOffset, lSize, &ParseMdiaAtom);
        default:
            return TRUE;
    }
}
int ParseEdtsAtom (HMMIO hmmio, long lName, long lOffset, long lSize) {
    switch (lName) {
        case mmioFOURCC ('e','l','s','t'):
            /* your code */
            return TRUE;
        default:
            return TRUE;
    }
}
int ParseMdiaAtom (HMMIO hmmio, long lName, long lOffset, long lSize) {
    switch (lName) {
        case mmioFOURCC ('m','d','h','d'):
            /* your code */
            return TRUE;
        case mmioFOURCC ('h','d','l','r'):
            /* your code */
            return TRUE;
        case mmioFOURCC ('m','i','n','f'):
            return CollectAtomsFromFile (hmmio, lOffset, lSize, &ParseMinfAtom);
        default:
            return TRUE;
    }
}
int ParseMinfAtom (HMMIO hmmio, long lName, long lOffset, long lSize) {
    switch (lName) {
        case mmioFOURCC ('s','t','b','l'):
            return CollectAtomsFromFile (hmmio, lOffset, lSize, &ParseStblAtom);
        default:
            return TRUE;
    }
}
int ParseStblAtom (HMMIO hmmio, long lName, long lOffset, long lSize) {
    switch (lName) {
        case mmioFOURCC ('s','t','s','d'):
            /* your code */
            return TRUE;
        case mmioFOURCC ('s','t','t','s'):
            /* your code */
            return TRUE;
        case mmioFOURCC ('s','t','s','s'):
            /* your code */
            return TRUE;
        case mmioFOURCC ('s','t','s','c'):
            /* your code */
            return TRUE;
        case mmioFOURCC ('s','t','s','z'):
            /* your code */
            return TRUE;
        case mmioFOURCC ('s','t','c','o'):
            /* your code */
            return TRUE;
        default:
            return TRUE;
    }
}
```

Listing Five

```
00000000    RIFF (00E5DE86) 'AVI '
0000000C        LIST (000007D4) 'hdrl'
00000018            avih (00000038)
                        TotalFrames  : 1270
                        Streams      : 2
                        InitialFrames: 8
                        MaxBytes     : 307200
                        BufferSize   : 30720
                        uSecPerFrame : 100000
                        Rate         : 10.000 fps
                        Size         : (320, 240)
                        Flags        : 0x00000710
                            AVIF_HASINDEX
                            AVIF_ISINTERLEAVED
                            AVIF_VARIABLESIZEREC
                            AVIF_NOPADDING
00000058            LIST (00000074) 'strl'
00000064                strh (00000038)
                            Stream Type   : vids
                            Stream Handler: cvid
                            Samp/Sec      : 10.000
                            Priority      : 0
                            InitialFrames : 0
                            Start         : 0
                            Length        : 1270
                            Length (sec)  : 127.0
                            Flags         : 0x00000000
                            BufferSize    : 14654
                            Quality       : 7500
                            SampleSize    : 0
000000A4                strf (00000028)
                            Size          : (320, 240)
                            Bit Depth     : 24
                            Colors used   : 0
                            Compression   : cvid
000000D4            LIST (0000005C) 'strl'
000000E0                strh (00000038)
                            Stream Type   : auds
                            Stream Handler: <default>
                            Samp/Sec      : 11025.000
                            Priority      : 0
                            InitialFrames : 8
                            Start         : 0
                            Length        : 139947
                            Length (sec)  : 126.9
                            Flags         : 0x00000000
                            BufferSize    : 1103
                            Quality       : 7500
                            SampleSize    : 1
00000120                strf (00000010)
                            wFormatTag    : WAVE_FORMAT_PCM
                            nChannels     : 1
                            nSamplesPerSec : 11025
                            nAvgBytesPerSec : 11025
                            nBlockAlign   : 1
                            nBitsPerSample : 8
00000138            vedt (00000008)
00000140        LIST (00E4E7F6) 'movi'
0000014C            LIST (0000045C) 'rec '
00000158                01wb (0000044F)
00000598            LIST (0000045A) 'rec '
000005A4                01wb (0000044E)
000009E0            LIST (0000045C) 'rec '
000009EC                01wb (0000044F)
00000E26            LIST (0000045A) 'rec '
00000E32                01wb (0000044E)
00001272            LIST (0000045C) 'rec '
0000127E                01wb (0000044F)
000016B8            LIST (0000045A) 'rec '
000016C4                01wb (0000044E)
00001AFE            LIST (0000045C) 'rec '
00001B0A                01wb (0000044F)
00001F44            LIST (0000045A) 'rec '
00001F50                01wb (0000044E)
0000238A            LIST (0000045C) 'rec '
00002396                01wb (0000044F)
000027D0            LIST (0000045A) 'rec '
000027DC                01wb (0000044E)
00002C16            LIST (00003A7E) 'rec '
00002C22                00dc (0000361A)
0000623C                01wb (0000044F)
00006676            LIST (000034A8) 'rec '
00006682                00dc (00003045)
00009AB8                01wb (0000044E)
00009AC4            ...
00E4E9CA            LIST (00000614) 'rec '
00E4E9D6                00dc (00000608)
00E4EFE6        idx1 (0000EEA0)
00E5DE8E
```

Listing Six

```
void ParseAVIMovie (char *szFileName) {
    MMCKINFO ckAVI, ckAVIH, ckHDRL, ckSTRL, ckSTRH, ckSTRF, ckIDX1, ckMOVI;
    MainAVIHeader avihdr;
    AVIStreamHeader strhdr;
    AVIINDEXENTRY avindx;
    HMMIO hmmio;
    long lStream;

    // Open file
    hmmio = mmioOpen (szFileName, NULL, MMIO_READ);

    // Read the AVI header
    mmioSeek (hmmio, 0, SEEK_SET);
    ckAVI.ckid = ckidRIFF;
    ckAVI.fccType = formtypeAVI;
    mmioDescend (hmmio, &ckAVI, 0, MMIO_FINDRIFF);
    ckHDRL.ckid = ckidLIST;
    ckHDRL.fccType = listtypeAVIHEADER;
    mmioDescend (hmmio, &ckHDRL, &ckAVI, MMIO_FINDLIST);
    ckAVIH.ckid = ckidAVIMAINHDR;
    mmioDescend (hmmio, &ckAVIH, &ckHDRL, MMIO_FINDCHUNK);
    mmioRead (hmmio, (HPSTR) &avihdr, sizeof(MainAVIHeader));

    // Read each stream header
    for (lStream = 0; lStream < (long) avihdr.dwStreams; lStream++) {
        ckSTRL.ckid = ckidLIST;
        ckSTRL.fccType = listtypeSTREAMHEADER;
        mmioDescend (hmmio, &ckSTRL, &ckHDRL, MMIO_FINDLIST);
        ckSTRH.ckid = ckidSTREAMHEADER;
        mmioDescend (hmmio, &ckSTRH, &ckSTRL, MMIO_FINDCHUNK);
        mmioRead (hmmio, (HPSTR) &strhdr, sizeof(AVIStreamHeader));
        mmioAscend (hmmio, &ckSTRH, 0);

        // Is it video?
        if (strhdr.fccType == streamtypeVIDEO) {
            /* your code */
        }

        // Or is it sound?
        else if (strhdr.fccType == streamtypeAUDIO) {
            /* your code */
        }
        // Loop until all streams processed
        mmioAscend (hmmio, &ckSTRL, 0);
    }

    // Done reading headers
    mmioAscend (hmmio, &ckHDRL, 0);
    mmioAscend (hmmio, &ckAVI, 0);

    // Locate movi data
    mmioSeek (hmmio, 0, SEEK_SET);
    ckAVI.ckid = ckidRIFF;
    ckAVI.fccType = formtypeAVI;
    mmioDescend (hmmio, &ckAVI, 0, MMIO_FINDRIFF);
    ckMOVI.ckid = ckidLIST;
    ckMOVI.fccType = listtypeAVIMOVIE;
    mmioDescend (hmmio, &ckMOVI, &ckAVI, MMIO_FINDLIST);
       /* your code */
    mmioAscend (hmmio, &ckMOVI, 0);
    mmioAscend (hmmio, &ckAVI, 0);

    // Locate index
    mmioSeek (hmmio, 0, SEEK_SET);
    ckAVI.ckid = ckidRIFF;
    ckAVI.fccType = formtypeAVI;
    mmioDescend (hmmio, &ckAVI, 0, MMIO_FINDRIFF);
    ckIDX1.ckid = ckidAVINEWINDEX;
    mmioDescend (hmmio, &ckIDX1, &ckAVI, MMIO_FINDCHUNK);
       /* your code */
    mmioAscend (hmmio, &ckIDX1, 0);
    mmioAscend (hmmio, &ckAVI, 0);

    // Close file
    mmioClose (hmmio);
}
```

End Listings

Compressing Waveform Audio Files

Cut your .WAV files in half with Windows' low-level waveform API

Neil G. Rowland, Jr.

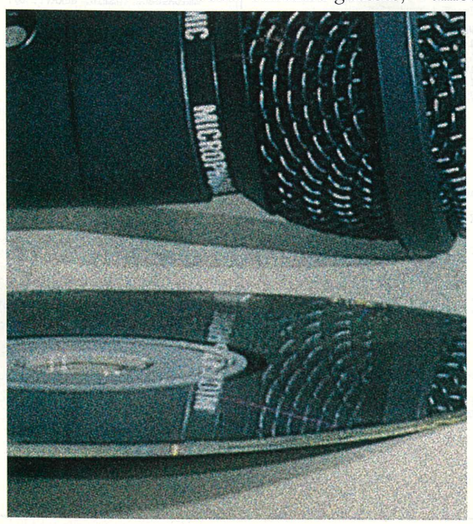
If you're a Windows multimedia developer and have been working with waveform audio files, you know that sampled waveform data can grow in size pretty fast. In fact, a single channel with a minimal sampling rate of 11.025 kHz at eight bits per sample translates into 11,025 bytes per second, or roughly 660 Kbytes per minute of audio data. Higher sampling rates and stereo each double these storage requirements.

Recently, I developed an application in C++, that uses the low-level waveform services to do some signal processing. It also happens to do something interesting. It compresses waveform files to about half the original size.

In this article, I'll show you how I accomplished this, and provide a class library that tames the Windows waveform API. To compile and run the source code from this article, you'll need a C++ compiler and the Microsoft Multimedia Development Kit (MDK). Of course, you'll also need a sound card with Windows drivers.

NYB1 Compression

The NYB1 format is a "lossy" compression scheme that I've developed. That is to say, it "loses" (throws out) some of the information in the input file in order to save space. The trick of it is to throw out the least-important information. NYB1 uses a single nybble for each sample, plus some slight overhead, and minus some compression of silence. If the input file uses one byte per sample (the usual case), then there is about 50 percent compression, guaranteed. I used some educated guesses, plus a little trial and error, to decide what should go in that nybble. The most obvious thing to try is just to store the top nybble of each sample. You still get a representation of the waveform, but it's less precise. In practice, this leads to a great deal of hiss in the output. This quantization noise is always present in digital audio, but is normally unobtrusive. Clearly, four bits per sample is not enough to keep it unobtrusive, at least by this brute-force method.



It's possible to improve the performance in the general case by storing the first-order differential of the input signal, then reintegrating it upon playback. Something similar to this goes on in analog tape recording. This works because of the way sound energy is distributed in the real world. Each octave in the audible range has roughly the same amount of sound on average as every other. (By roughly, I mean within an order of magnitude.) Human hearing seems constructed to take advantage of this. So is audio equipment. Look at any graphic equalizer with a spectrum analyzer. The bands are divided into equal sections of a logarithmic scale of frequency, usually by octaves. The spectrum analyzer outputs pink noise, which has an exactly equal distribution of energy per octave. When you view pink noise on the spectrum analyzer, it looks like a flat line.

But the brute-force method of digital coding, where you just convert each sample into a linear number, works best for a very different situation. The most complex waveform you can throw at it is a completely random value for each sample. This amounts to what's called "white noise." White noise has its energy equally distributed across frequencies on a linear frequency scale (not a logarithmic one), which is very rare in nature.

When you store the sort of sounds you hear in nature this way, all but the highest frequencies are stored inefficiently. Because the amplitudes of the lower frequencies are typically much lower than those of the higher frequencies, a scaling factor that does justice to the high frequencies makes the low frequencies use only part of the range of numbers. Thus, the low-frequency sounds are stored with less than the full precision the sample has room for. This translates into a poorer signal-to-noise ratio in the lower frequencies. (The "noise" in this signal-to-noise is quantization noise.)

You can turn pink noise into white noise by taking its first-order differential. Do the same to an audio waveform, and you've translated something similar to pink noise into something similar to white noise. Store this quasi-white noise digitally, and you're making the most of the digital medium. You've adapted the nature of the input to map to the medium's strengths.

During less-loud parts of the waveform, the quantization noise can swamp the "real" waveform because there isn't enough precision in the sample to represent small (soft) details. Even with the first-order differential, this is a problem. One solution is to change the scale of samples to reflect the range of the waveform values. That way, precision isn't wasted.

I've broken the sample into two parts, a mantissa and an exponent. The mantissa is the sample proper. It represents the first-order differential of the waveform, to a certain scale. Each mantissa is one nybble. The exponent indicates the scale. I call the exponent "shrite," short for "shift right." On encoding, I take the first-order differential as a 16-bit integer, then I shift it right by the exponent. This gives me a mantissa in the bottom four bits that I can store. On decoding, I do the reverse.

It takes one nybble for the mantissa and one nybble (roughly) for the exponent. This adds up to one byte, the same size as a typical, uncompressed .WAV file sample. Obviously, you gain nothing if you store the exponent with each mantissa. Therefore, I divide the incoming waveform samples into groups of seven each. I take their first-order differentials and determine the exponent that is appropriate for the highest amplitude sample in the group. This will be the shrite value for the entire group. Thus, I store four bytes (seven mantissa nybbles plus one exponent nybble) for each seven samples in the original waveform. This is reasonably close to 2:1 compression.

The number seven is somewhat arbitrary. There's a trade-off between precision and compression ratio. If you make the group smaller, then there are more exponents (shrites) in the file, and the storage efficiency suffers. If you make the group larger, then the one shrite chosen for the greatest-magnitude sample is less likely to be well suited for any other sample in the group. Consider the extreme case, where there's one exponent for all the mantissas, which is the same as having no exponent at all. It's a case of one size fits one, and can be made to do for a few, but not for many.

Finally, I compress silent sections by leaving out the mantissas altogether. I simply store the exponent of each group, which is 0.

Encoding/Decoding Logic

The code to work with WAV files is in WAVELIB.CPP; see Listing Two (page 24). The prototypes are in WAVELIB.H; Listing One (page 24). The WAV library has three classes: WAVUSR, which is a superclass; WAVRDR to read WAV files; and WAVWRT to write them. WAVUSR's base class is RIFFUSR, which contains logic common to RIFF files. Both WAVRDR and WAVWRT have APIs that let the caller deal with them on a sample-by-sample basis, and all the messy details of buffering are hidden inside the classes. The classes also partially hide the differences between mono and stereo and the detail of how many bits each sample consists of.

At any given point, an open WAV object has a file position that is on a given sample. When you read a sample via *mNextSample()* or write a sample via *mWriteSample()*, the object steps to the next sample in the file. Each class has three public members named *sample*, *left*, and *right*. These unsigned 16-bit values hold the current sample. The value 32,768 represents an amplitude of 0. *Left* and *right* are the current values for the left and right channel. *Sample* is the average of the two. Each *mNextSample()* reads in the next sample from the WAV file and fills in these three values accordingly. *mWriteSample()* writes out *sample* if it's a mono file, or *left* and *right* if it's stereo.

The NYB1 library (see Listing Three, page 25 and Listing Four, page 26) shares code with the WAV library by means of the RIFFUSR class. It contains the classes NYBUSR, NYBRDR, and NYBWRT. These all have interfaces identical to the corresponding WAV classes. So the caller can treat them as if they were WAV classes. The process of encoding and decoding in the NYB1 format is hidden inside these classes.

The NYBWRT version of *mWriteSample()* collects incoming samples into groups. When it's just been called for the last sample in a group, it processes the group it's built. First, it figures the exponent by taking a first-order differential of all the samples, figures the appropriate shrite value for the largest differential it sees, and calls *iNegFeedbackStage()* for each mantissa in the group. *iNegFeedbackStage()* does three things: It applies the encoding by calling *iEncode()*; writes the encoded group out to the NYB1 file; and decodes each encoded mantissa and compares the result to the original value. *iNegFeedbackStage()* determines the sign of the error value and on the next sample applies this as a bias to the encoded sample before writing it out. This has the effect of preventing cumulative error, by a sort of negative feedback.

The decoding is done by the NYBRDR version of *mNextSample()*, which reads in nybbles from the NYB1 file, stores shrite values when it encounters them, and decodes mantissas. There's no need to worry about cumulative error at this phase. The *iNegFeedbackStage()* routine acted to prevent this when the file was created. Finally, *mNextSample()* recognizes when you're in a zero-compressed group and dummies up seven silent samples (sample=32,768).

Using the Library

The sample application that I've provided demonstrates how to use the NYB1 library. *WinMain()* in Listing Five (page 27) simply shovels samples between the input and output objects, one sample at a time. On conversion, it opens the input WAV file by means of WAVRDR object, and the output NYB1 file by means of a NYBWRT object. Then it repeatedly calls *mNextSample()*, feeds the sample to the NYBWRT object, and calls an *mWriteSample()* for that object. On playback, the input object is a NYBRDR, and the output object is a WAVEPLAYER. Again, it's a matter of ferrying samples from one to the other, one at a time, until there are no more samples at the input end.

When running the program, notice that the main screen is a small dialog with two edit fields. The top field is for an input wave file. If you want to convert a file from WAV to NYB1 format, enter the full path and file specification of the source file here. To simply play

Neil is a programmer at Gradient Technologies, porting OSF's Distributed Computing Environment to DOS/Windows machines. He can be reached on CompuServe at 72133,426, on the Internet as neil_r@gradient.com, and on Channel One as "Neil Rowland."

an existing NYB1 file, leave this blank. The second edit control takes the full path and file specification of the NYB1 file. If you leave it blank, it defaults to \X.NYB. For playback, this is the input specification. For conversion, it is the output specification.

The WAVEPLAYER Class

The output stage is encapsulated in the WAVEPLAYER class (see Listing Two). The waveform portion of a sound-card driver for Windows takes its input in buffers. Then it plays the current buffer in the background while the CPU prepares the next buffer for it. This is good for efficiency, but you'll want to deal with the waveform output as a serial string of samples. The Windows API does almost nothing to hide the details of buffering from us. So, I do it in the WAVEPLAYER class. The programmer using the class feeds it a string of samples, one at a time. The WAVEPLAYER object handles the buffering. The code calling WAVEPLAYER need not bother with any part of the Windows waveform API calls.

When WAVEPLAYER is started, it opens the wave-playing side of the sound card. The *WAVEPLAYER::mOpen()* method uses Windows API call *waveOutOpen()* to open a WAVE_MAPPER device, which specifies a default device for waveform handling. Currently, it's the only device ID supported by Windows. The *mOpen()* method then allocates a buffer header, of type WAVEHDR. This is the Windows object that manages the buffers for the sound card. When *WAVEPLAYER::mPlaySample()* is called for a sample, it's usually appended to the samples in the current buffer. But when there's no room in the buffer, it must flush the buffer. *iCloseoutSampleBuffer()* contains the logic to flush the previous buffer. The call to *waveOutUnPrepareHeader()* tells Windows you're done with a buffer. It won't return (successfully) unless and until Windows itself is done playing the contents of the buffer. When *waveOutUnPrepareHeader()* returns you can safely free the old buffer.

A note on the parameter to *WAVEPLAYER::mOpen* is needed. Though its type is a pointer to type WAVEFORMAT, it is really being passed a PCMWAVEFORMAT. This oddity is a result of the way the Windows API call *waveOutOpen* is prototyped. This call takes a pointer to type WAVEFORMAT. But in reality, when the device type in the passed structure is WAVE_FORMAT _PCM (and it always is), it should point to a PCMWAVEFORMAT. The difference is that PCWAVEFORMAT has an extra field, *wBitsPerSample,* that Windows must have. In NYBI.CPP, PlayIt makes the call *Play.mOpen (&NybIn.Fmt.wf)*. This is the same as saying *Play.mOpen ((WAVEFORMAT*)(&NybIn.Fmt))*.

mPlaySample() "hands" Windows the buffer just filled, calling the *iPlay()* method to handle the mechanics of handing over a buffer. *iPlay()* sets up the WAVHDR to point to the new buffer, and calls *waveOutPrepareHeader()* and *waveOutWrite()*. Both calls are necessary to tell Windows to play the buffer.

iCloseoutSampleBuffer() then sets the buffer pointer to NULL and returns. The next call to *mPlaySample()* will see this null pointer, allocate a new buffer, and start filling it. For now, *mPlaySample()* simply returns to the caller.

When you've finished with the output, and closed the WAVEPLAYER, flush the current buffer, even though it's probably not full. Otherwise, you'll never hear the last bit of the waveform. *iCloseoutBuffer()* is the method that does this. First it waits until the sound card is done with the previous buffer. The WHDR_DONE flag in the WAVEHDR that manages the buffer will go to 0 when this happens. Then it executes a final *waveOutUnPrepareHeader()* to tell Windows you're done with the final buffer. Finally, it frees the buffer and returns.

RIFFUSR and Friends

Windows Multimedia file formats are all special cases of the RIFF file format. Therefore, a WAV file is a type of RIFF file. For consistency, I've decided to make my NYB1 file format a RIFF format as well. Windows has an API to help parse RIFF files. Both the WAV and NYB1 libraries need it, so I've encapsulated it in the RIFFUSR class.

A RIFF file consists of "chunks," each of which has a 4-byte type, a length field, and the actual data. These chunks, which vary according to format, can be nested. In the case of a WAV file, there is one top-level RIFF chunk of type WAVE, containing a format chunk (type *fmt*) and a large data chunk (type *data*) that takes up most of the file. I use the same scheme in NYB1 format to help share code. The only difference is that the top-level chunk is of type NYB1. The RIFFUSR methods *iOpenRead()* and *iOpenWrite()* are made possible by this consistency. *iOpenRead()* and *iOpenWrite()* use the Windows call *mmioOpen()* to open the file in such a way that it can be manipulated with the *mmio*()* calls. The option flag MMIO_ALLOCBUF instructs Windows to allocate buffers for use.

The Windows API for RIFF files deals with these chunks almost as if they were files. There are calls to open, close, read, and write chunks. The API calls *mmioAscend()* and *mmioDescend()* determine which chunk within the file is currently open. *mmioDescend()* moves the read pointer into a chunk within the current chunk. If there's not currently one, it moves into a top-level chunk. It finds the chunk, given its name in the parameter *ck*, and then moves the file pointer to point to the first data byte of that chunk, essentially opening it. Note that since chunks can be nested, so can *mmioDescend()* calls. *mmioRead()* and *mmioWrite()* are analogous to the standard library *read()* and *write()* calls, except that they deal with the current chunk instead of the whole file. *mmioRead()* won't read past the end of the chunk. *mmioWrite()* appends to the current chunk.

Finally, *mmioAscend()* is analogous to *close()*. On writing, it closes out the chunk in a tidy manner. On reading, it tells Windows that you are done reading that chunk. You have then "ascended" one level higher in the hierarchy of nested chunks. You can then do another *mmioDescend()* to select another chunk to work with. Also note that because *mmioDescend()* calls can be nested, *mmioAscend()* calls are also nested. Every *mmioDescend()* must have a matching *mmioAscend()*.

This API is somewhat object oriented. All of the calls with the "mmio" prefix take a handle of type HMMIO as the first parameter. HMMIO is a scalar that acts like a file handle, but it is especially for RIFF files and for the mmio calls. There's also an MMIOINFO structure, which holds various useful bits of information about the RIFF file. The most important items in MMIOINFO are the next and end pointers to the current buffer (fields *pchNext* and *pchEndWrite*). Though I use *mmioSetBuffer()* to tell Windows to allocate buffers for me, I still need to know where they are since I must manipulate them directly. The call *mmioGetInfo()* reads the MMIOINFO data. I store both the HMMIO handle and the MMIOINFO data in the RIFFUSR object.

The inline functions *iReadByte()* and *iWriteByte()* encapsulate this buffer business. Using these calls, I can write my code as if I were dealing with a stream of bytes, instead of a buffer-oriented API. The calls are in WAVELIB.H. When they need to go to another buffer, they call *iReadBuf()* and *iWriteBuf()*. These routines use the Windows API call *mmioAdvance()* to get another buffer and simultaneously update your copy of the MMIOINFO data. One messy detail: On writing, you must set the MMIO_DIRTY flag in MMIOINFO so that Windows knows that you want this buffer written out. Then you must do *mmioSetInfo()* so that Windows' internal information is in sync with yours and knows you've set the flag.

Finally, when you close an output RIFF file, the last buffer needs to be flushed. Consider *RIFFUSR::iClose()* in Listing Two. First set MMIO_DIRTY and do the *mmioSetBuffer()* to tell Windows to flush the buffer. However, Windows doesn't flush it just then. Do an *mmioAscend()*, which writes the last buffer out to disk, and takes the file pointer out of the chunk. But you're still in the RIFF chunk (recall that these chunks are nested). One more *mmioAscend()* to get out of this, and finally an *mmioClose()* to tell Windows that you're done with the RIFF file.

Future Enhancements

NYB1 is not the final word on compressing waveforms cheaply and easily, but it's a good starting point, and many areas can be improved or enhanced. I've left out certain optimizations for clarity's sake. For example, you may want to rewrite *iNegFeedback()* so it no longer does the differential twice for each sample.

There are also possibilities for enhancements in the format itself. For example, you may want to use more bits for louder sections. When the shrite value is high, the quantization noise that isn't filtered out by various tricks is very noticeable. To counter this, use more than four bits for each mantissa. Experiment to determine just how much more and where the cutoff point should be. Remember, you're sacrificing compression here for greater clarity.

Also, consider the case of 16-bit input samples. Should NYB2 be able to creditably handle high-fidelity waveforms? This could call for a different trade-off. Consider a separate formula for increasing the bits per mantissa in the case of a 16-bit input. Remember, the header of the .NYB file contains a copy of the header of the original WAV file, so you'll know which formula to use on playback.

A final enhancement might be to provide for stereo. This is simple enough. If the header says the input was stereo, then use left/right channel pairs of mantissas. The exponent (shrite) needn't be doubled in this way. Even when the overall loudness varies between left and right channels, the listener probably won't notice any improvement in clarity in the quieter channel, because the louder channel will drown it out.

DDJ

(Listings begin on page 24.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

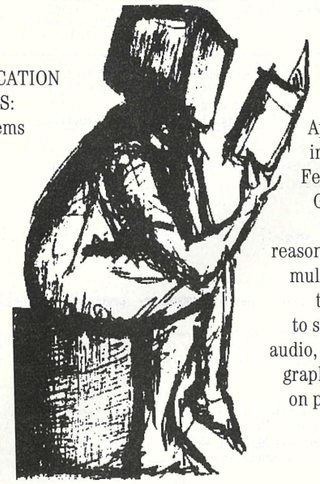
Indexes of
Dr. Dobb's Journal
1982–1992

- Cumulative Subject and Author Indexes covering in detail the years 1982–1992, bound in *ONE* volume
- 8"×11" paperback with 120 pages and 7900 entries
- $21.95 U.S. ppd
- Epson printer files available on disk: $14.95 U.S. ppd

Call: 804-977-7015
Fax: 804-286-4744

Stephen Bach
1411 A Short 18th Street
Charlottesville, VA 22902-5402

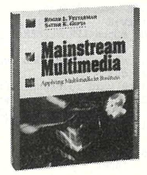
VNR KNOWS MULTIMEDIA



NEW!

VOICE COMMUNICATION WITH COMPUTERS: Conversation Systems by Christopher Schmandt. How to make computers talk, recognize and respond to verbal commands. Covers desktop audio, computers, and phones and much more.
$39.95 0-442-23935-1

MAINSTREAM MULTIMEDIA: Applying Multimedia in Business by Roger Fetterman and Satish Gupta. Covers costs, uses, training, and reasons for implementing multimedia in business today. Explains how to seamlessly integrate audio, images, video, text, graphics, and animation on personal computers.
$39.95 0-442-01181-4



Van Nostrand Reinhold
115 Fifth Avenue New York, New York 10003

To place an order, or for a complete listing of VNR titles, call 1-800-544-0550 or GO VNR on the CompuServe® Network.

CIRCLE NO. 763 ON READER SERVICE CARD

Listing One *(Text begins on page 20.)*

```
//***************************** WAVELIB.H ***********************************
// Class library for Windows waveforms and MIDI.
// Copyright (c) 1993 by  Neil G. Rowland, Jr. Ø4-JUN-93
#ifndef __WAVELIB_H
#define __WAVELIB_H
extern "C"
    {
    #include <windows.h>
    #include <mmsystem.h>
    }
#pragma hdrstop
//***************************************************************************

class RIFFUSR
    { // Base class for a user for a RIFF file.
  public:
    HMMIO       hmmio;      // handle to open WAVE file
    MMCKINFO    ckRIFF;     // chunk info. for RIFF chunk
    MMCKINFO    ck;         // info. for a chunk
    MMIOINFO    mmioinfo;   // current status

    RIFFUSR()  { hmmio = NULL; };

  protected:
    BOOL iOpenRead(char* _pszInFile, char* _pFmt, int _fmtlen,
                                        char _t1, char _t2, char _t3, char _t4);
    void iCloseRead();
    BOOL iReadBuf();
    BOOL iReadByte(BYTE& _byte)
        { // If we are at end of the input file I/O buffer, fill it.
          // Test that we don't hit end of file while (lSamples > Ø).
            if (mmioinfo.pchNext == mmioinfo.pchEndRead)  {
            if (!iReadBuf())  { MessageBeep(Ø);  return FALSE; }
            if (mmioinfo.pchNext == mmioinfo.pchEndRead) return FALSE;
        }
        _byte = *(mmioinfo.pchNext);
        mmioinfo.pchNext++;
        return TRUE;
        };
    BOOL iOpenWrite(char* _pszOutFile, const char* _pFmt, int _fmtlen,
                    char _t1, char _t2, char _t3, char _t4);
    void iCloseWrite();
    BOOL iWriteBuf();
    BOOL iWriteByte(BYTE _byte)
        {
        if (!mmioinfo.pchNext)  return FALSE;
        if (mmioinfo.pchNext >= mmioinfo.pchEndWrite)
            { // If at end of output file I/O buffer, flush it.
                if (!iWriteBuf())
                    { MessageBeep(Ø);  return FALSE; }
            }
        if (!mmioinfo.pchNext)  { MessageBeep(Ø);  return FALSE; }
        *mmioinfo.pchNext = _byte;
        mmioinfo.pchNext++;
        return TRUE;
        };
    };
//---------------------------------------------------------------------------
class WAVUSR : public RIFFUSR
    { // User for a .WAV file.
  public:
        PCMWAVEFORMAT Fmt;      // format of WAVE file.
    WORD    sample, left, right;    // average, left and right channels.
    };
class WAVRDR : public WAVUSR
    {
  public:
    BOOL mOpenRead(char* _pszInFile);
    BOOL mNextSample();
    void mClose();
  protected:
    BOOL iReadChanSample(WORD& _sample);
    };
class WAVWRT : public WAVUSR
    {
  public:
    BOOL mOpenWrite(char* _pszOutFile, const PCMWAVEFORMAT* _pFmt);
    BOOL mWriteSample();
    void mClose();
  protected:
    BOOL iWriteChanSample(WORD _sample);
    };
typedef struct
    { // Play waveform output
    HWAVEOUT    hwaveout;
    HANDLE      hHdr;
    LPWAVEHDR   lpHdr;
    HANDLE      hBuf;       // current waveform buffer.
    // accum buffer for feeding in a sample at a time...
    HANDLE      hBufS;
    LPSTR       lpBufS;
    unsigned    countS;
    BOOL mOpen(LPWAVEFORMAT _pFmt);
    void mClose();
    void mPlaySample(WORD _sample);
  protected:
    inline void iCloseoutBuffer();
    inline void iCloseoutSampleBuffer();
    inline void iPlay(MMIOINFO* _pInfo);
    inline void iPlay(HANDLE _hbuf, LPSTR _lpBuf, int _len);
    }
WAVEPLAYER;
#endif  //ndef __WAVELIB_H
```

Listing Two

```
//***************************** WAVELIB.CPP *********************************
// Class library for Windows waveforms.
// Copyright (c) 1993 by Neil G. Rowland, Jr. Ø4-JUN-93
#include "wavelib.h"
//***************************************************************************

void lmemcpy(LPSTR _pszDest, LPSTR _pszSrc, DWORD _len)
    {
    if (!_pszDest)  return;
    if (!_pszSrc)  return;
    if (!_len)  return;
    while (_len--)  {
        *(_pszDest++) = *_pszSrc;
        _pszSrc++;
        }
    }
//***************************** RIFFUSR *************************************
#define BUFSIZE 25ØØ
BOOL RIFFUSR::iOpenRead(char* _pszInFile, char* _pFmt, int _fmtlen,
                        char _t1, char _t2, char _t3, char _t4)
    { // open a generic RIFF file for reading.
      // on success, it is descended into the data chunk,
      // and *_pFmt holds a copy of the fmt chunk.
    hmmio = mmioOpen(_pszInFile, NULL, MMIO_ALLOCBUF ¦ MMIO_READ);
    if (hmmio == NULL)  return FALSE;     // cannot open RIFF file

    mmioSetBuffer(hmmio, NULL, BUFSIZE, Ø); // allocate buffers
    // Descend the input file into the 'RIFF' chunk.
    if (mmioDescend(hmmio, &ckRIFF, NULL, Ø) != Ø)
        goto ERROR_BAD_FORMAT;
    // Make sure the input file is of the desired type....
    if ((ckRIFF.ckid != FOURCC_RIFF) ¦¦
        (ckRIFF.fccType != mmioFOURCC(_t1, _t2, _t3, _t4)))
        goto ERROR_BAD_FORMAT;
    // Search the input file for for the 'fmt ' chunk.
    ck.ckid = mmioFOURCC('f', 'm', 't', ' ');
    if (mmioDescend(hmmio, &ck, &ckRIFF, MMIO_FINDCHUNK) != Ø)
        goto ERROR_BAD_FORMAT;      // no 'fmt ' chunk
    // Expect the 'fmt' chunk to be at least as large as _fmtlen;
    // if there are extra parameters at the end, we'll ignore them
    if (ck.cksize < (long) _fmtlen)
        goto ERROR_BAD_FORMAT;      // 'fmt ' chunk too small
    // Read the 'fmt ' chunk into *_pFmt.
    if (mmioRead(hmmio, (HPSTR) _pFmt, (long)_fmtlen) != (long)_fmtlen)
        return FALSE;     // truncated file, probably
    // Ascend the input file out of the 'fmt ' chunk.
    if (mmioAscend(hmmio, &ck, Ø) != Ø)  return FALSE; // truncated file?
    // Search the input file for for the 'data' chunk, and descend.
    ck.ckid = mmioFOURCC('d', 'a', 't', 'a');
    if (mmioDescend(hmmio, &ck, &ckRIFF, MMIO_FINDCHUNK) != Ø)
        goto ERROR_BAD_FORMAT;      // no 'data' chunk
    mmioGetInfo(hmmio, &mmioinfo, Ø);
    return TRUE;
  ERROR_BAD_FORMAT:
    MessageBox(NULL,"Input file must be a RIFF file", "RIFFUSR::mOpenRead",
                                                    MB_ICONEXCLAMATION¦ MB_OK);
    iCloseRead();
    return FALSE;
    }
void RIFFUSR::iCloseRead()
    {
    if (!hmmio)  return;
    mmioSetInfo(hmmio, &mmioinfo, Ø);   // properly close out input file.
    mmioClose(hmmio, Ø);
    hmmio = NULL;
    }
BOOL RIFFUSR::iReadBuf()
    { return mmioAdvance(hmmio, &mmioinfo, MMIO_READ) == Ø; }
//---------------------------------------------------------------------------
BOOL RIFFUSR::iOpenWrite(char* _pszOutFile, const char* _pFmt, int _fmtlen,
                         char _t1, char _t2, char _t3, char _t4)
    {
    // Open the output file for writing using buffered I/O. Note that
    // if the file exists, the MMIO_CREATE flag causes it to be truncated
    // to zero length.
    hmmio = mmioOpen(_pszOutFile, NULL, MMIO_ALLOCBUF¦ MMIO_WRITE¦
                                                        MMIO_CREATE);
    if (hmmio == NULL)  return FALSE;    // cannot open WAVE file

    mmioSetBuffer(hmmio, NULL, BUFSIZE, Ø); // allocate buffers
    // Create the output file RIFF chunk of desired type.
    ckRIFF.fccType = mmioFOURCC(_t1, _t2, _t3, _t4);
    if (mmioCreateChunk(hmmio, &ckRIFF, MMIO_CREATERIFF) != Ø)
        goto cantwrite;    // cannot write file, probably
    // We are now descended into the 'RIFF' chunk we just created.
    // Now create the 'fmt ' chunk. Since we know the size of this chunk,
    // specify it in the MMCKINFO structure so MMIO doesn't have to seek
    // back and set the chunk size after ascending from the chunk.
    ck.ckid = mmioFOURCC('f', 'm', 't', ' ');
    ck.cksize = _fmtlen;    // we know the size of this ck.
    if (mmioCreateChunk(hmmio, &ck, Ø) != Ø)  goto cantwrite;
    // Write the *_pFmt data to the 'fmt ' chunk.
    if (mmioWrite(hmmio, (HPSTR) _pFmt, _fmtlen) != _fmtlen)
        goto cantwrite;
    // Ascend out of the 'fmt ' chunk, back into the 'RIFF' chunk.
    if (mmioAscend(hmmio, &ck, Ø) != Ø)  goto cantwrite;
    // Create the 'data' chunk that holds the waveform samples.
    ck.ckid = mmioFOURCC('d', 'a', 't', 'a');
    if (mmioCreateChunk(hmmio, &ck, Ø) != Ø)  goto cantwrite;
    mmioGetInfo(hmmio, &mmioinfo, Ø);
    return TRUE;
  cantwrite:
    iCloseWrite();
    return FALSE;
    }
void RIFFUSR::iCloseWrite()
    {
    if (!hmmio)  return;
    // flush the output RIFF chunk...
    mmioinfo.dwFlags ¦= MMIO_DIRTY;
    if (mmioSetInfo(hmmio, &mmioinfo, Ø) != Ø)  goto ERROR_CANNOT_WRITE;
    // cannot flush, probably
    // Ascend the output file out of the 'data' chunk -- this will cause
    // the chunk size of the 'data' chunk to be written.
    if (mmioAscend(hmmio, &ck, Ø) != Ø)  goto ERROR_CANNOT_WRITE;
    // cannot write file, probably
    // Ascend the output file out of the 'RIFF' chunk -- this will cause
    // the chunk size of the 'RIFF' chunk to be written.
    if (mmioAscend(hmmio, &ckRIFF, Ø) != Ø)  goto ERROR_CANNOT_WRITE;
    // cannot write file, probably
    goto close;
  ERROR_CANNOT_WRITE:
    MessageBox(NULL, "Error closing out output file", "RIFFUSR::mCloseWrite",
MB_ICONEXCLAMATION¦ MB_OK);
  close:
    mmioClose(hmmio, Ø);
    hmmio = NULL;
    }
BOOL RIFFUSR::iWriteBuf()
    {
    mmioinfo.dwFlags ¦= MMIO_DIRTY;
    if (mmioAdvance(hmmio, &mmioinfo, MMIO_WRITE) != Ø)  return FALSE;
    return TRUE;
    }
//***************************************************************************
BOOL WAVRDR::mOpenRead(char* _pszInFile)
    {
    if (!RIFFUSR::iOpenRead(_pszInFile, (char*)&Fmt,
                                        sizeof(PCMWAVEFORMAT), 'W', 'A', 'V', 'E'))
        return FALSE;
    // Make sure the input file is a PCM WAVE file of a variety we support.
    if ((Fmt.wf.wFormatTag != WAVE_FORMAT_PCM))  goto ERROR_BAD_FORMAT;
    // bad input file format
    if ((Fmt.wBitsPerSample != 8) && (Fmt.wBitsPerSample != 16))
        goto ERROR_BAD_FORMAT;      // bad input file format
    return TRUE;
  ERROR_BAD_FORMAT:
    MessageBox(NULL, "Input file must be a PCM WAVE file",
                                "WAVUSR::mOpenRead", MB_ICONEXCLAMATION¦ MB_OK);
  ERROR_BAD_FORMAT1:
    mClose();
    return FALSE;
    }
BOOL WAVRDR::iReadChanSample(WORD& _sample)
    { // read one sample for one channel.
    BYTE    c;
    if (!iReadByte(c))  return FALSE;
    _sample = c << 8;
    if (Fmt.wBitsPerSample > 8)  { // 16 bit
        if (!iReadByte(c))  return FALSE;
        _sample ¦= c;
        }
    return TRUE;
    }
BOOL WAVRDR::mNextSample()
    { // read the next sample into fields sample, left and right.
    if (Fmt.wf.nChannels == 1)  { // mono
        if (!iReadChanSample(sample))  return FALSE;
        left = right = sample;
        }
    else  { // stereo
        if (!iReadChanSample(left))  return FALSE;
        if (!iReadChanSample(right))  return FALSE;
        sample = left>>1 + right >>1;
        }
    return TRUE;
    }
void WAVRDR::mClose()
    { RIFFUSR::iCloseRead(); }
//***************************************************************************
BOOL WAVWRT::mOpenWrite(char* _pszOutFile, const PCMWAVEFORMAT* _pFmt)
    {
    if (!RIFFUSR::iOpenWrite(_pszOutFile, (const char*)_pFmt,
    sizeof(PCMWAVEFORMAT), 'W', 'A', 'V', 'E'))
        return FALSE;
    Fmt = *_pFmt;
    return TRUE;
  cantwrite:
    mClose();
    return FALSE;
    }
void WAVWRT::mClose()
    { RIFFUSR::iCloseWrite(); }
BOOL WAVWRT::iWriteChanSample(WORD _sample)
    {
    if (mmioinfo.pchNext >= mmioinfo.pchEndWrite-1)  {
    // If we are at the end of the output file I/O buffer, flush it.
        if (!iWriteBuf())  { MessageBeep(Ø);  return FALSE; }
        }
    *(mmioinfo.pchNext)++ = HIBYTE(_sample);
    if (Fmt.wBitsPerSample > 8)  // 16 bit
        *(mmioinfo.pchNext)++ = LOBYTE(_sample);
    return TRUE;
    }
BOOL WAVWRT::mWriteSample()
    {
    if (Fmt.wf.nChannels == 1)  // mono
        if (!iWriteChanSample(sample))  return FALSE;
    else  { // stereo
        if (!iWriteChanSample(left))  return FALSE;
        if (!iWriteChanSample(right))  return FALSE;
        }
    return TRUE;
    }
//***************************************************************************
inline void WAVEPLAYER::iCloseoutBuffer()
    { // Flush and free the current buffer.
    if (!lpHdr ¦¦ !hBuf)  return;
    if (!(lpHdr->dwFlags&WHDR_PREPARED))  return;
    // Finish up with previous buffer...
    while ((lpHdr->dwFlags&WHDR_DONE) == Ø)  Yield();   // wait
    waveOutUnprepareHeader(hwaveout, lpHdr, sizeof(WAVEHDR));
    GlobalUnlock(hBuf);  GlobalFree(hBuf);
    hBuf = NULL;
    }
inline void WAVEPLAYER::iCloseoutSampleBuffer()
    { // flush buffer for sample mode...
    if (!hBufS ¦¦ !lpBufS)  return;
    if (countS)  iPlay(hBufS, lpBufS, countS); // no longer own old buffer.
    lpBufS = NULL;  // so will get new buffer.
    hBufS = NULL;
    }
inline void WAVEPLAYER::iPlay(MMIOINFO* _pInfo)
    {
    if (!hwaveout)  return;
    if (!_pInfo)  return;
    if (!lpHdr)  return;
    DWORD   len = _pInfo->pchNext - _pInfo->pchBuffer;
    HANDLE  hNewBuf = GlobalAlloc(GMEM_MOVEABLE¦ GMEM_SHARE, len);
    LPSTR   lpNewBuf = GlobalLock(hNewBuf);
    lmemcpy(lpNewBuf, (LPSTR)_pInfo->pchBuffer, len);
    iPlay(hNewBuf, lpNewBuf, len);
    }
inline void WAVEPLAYER::iPlay(HANDLE _hbuf, LPSTR _lpBuf, int _len)
    { // the passed buffer will be freed later, not by caller...
    iCloseoutBuffer();  // finish with previous.
    // Queue this buffer's worth...
    hBuf = _hbuf;
    lpHdr->lpData = _lpBuf;
    lpHdr->dwBufferLength = _len;
    lpHdr->dwLoops = ØL;
    lpHdr->dwFlags = ØL;    // MPG, p5-28
    if (Ø == waveOutPrepareHeader(hwaveout, lpHdr, sizeof(WAVEHDR)))
        waveOutWrite(hwaveout, lpHdr, sizeof(WAVEHDR));
    else MessageBeep(Ø);
    }
//---------------------------------------------------------------------------
BOOL WAVEPLAYER::mOpen(LPWAVEFORMAT _pFmt)
    {
    hwaveout = NULL;  hHdr = NULL;  lpHdr = NULL;
    hBufS = NULL;  lpBufS = NULL;  countS = Ø;
    if (!_pFmt)  return FALSE;
    if (Ø != waveOutOpen(&hwaveout, WAVE_MAPPER, _pFmt, NULL, NULL, Ø))
                                                            return FALSE;
    // allocate the buffer header, but no buffer yet...
    hBuf = NULL;
    hHdr = GlobalAlloc(GMEM_MOVEABLE¦ GMEM_SHARE, sizeof(WAVEHDR));
    lpHdr = (LPWAVEHDR)GlobalLock(hHdr);
    lpHdr->dwFlags = Ø;
    return TRUE;
    }
void WAVEPLAYER::mClose()
    {
    iCloseoutSampleBuffer();
    iCloseoutBuffer();
    if (hwaveout)  waveOutClose(hwaveout);
    if (hHdr)  { GlobalUnlock(hHdr);  GlobalFree(hHdr); }
    hwaveout = NULL;
    hHdr = NULL;  lpHdr = NULL;

    if (hBufS)  {
        GlobalUnlock(hBufS);  GlobalFree(hBufS);
        hBufS = NULL;  lpBufS = NULL;
        }
    }
void WAVEPLAYER::mPlaySample(WORD _sample)
    {
    if (!lpBufS)  { // need a new buffer...
        hBufS = GlobalAlloc(GMEM_FIXED, BUFSIZE+1);
        if (!hBufS)  return;
        lpBufS = GlobalLock(hBufS);
        countS = Ø;
        }
    lpBufS[countS] = HIBYTE(_sample);
    if (++countS >= BUFSIZE)
        iCloseoutSampleBuffer();
    }
```

Listing Three

```
//***************************** NYBLIB.H ************************************
// Main header file for NYB1 compressed waveform utility.
// Copyright (c) 1993 by Neil G. Rowland, Jr. 24-JUN-93

#include "wavelib.h"

//***************************************************************************
// Access to NYB1 files. Makes it seem like .WAV files
// A group has this many nybbles, preceded by a magnitude nybble...
#define SAMPLESPERGROUP 7
class NYBUSR : public RIFFUSR
    { // User for a .NYB file.
  public:
    PCMWAVEFORMAT Fmt;      // format of would-be WAVE file.
    WORD        sample;     // 16-bit .WAV style sample.
    NYBUSR();
  protected:
    int     groupcount;
    WORD    encprevsample;  // used by iEncode.
    WORD    decprevsample;  // used by iDecode.
    BYTE    shrite; // maginitude of current group (shift-right amount)
    inline void iDecode(WORD& _sample);
    inline void iEncode(WORD& _sample);
    };
class NYBRDR : public NYBUSR
    {
  public:
    unsigned samplespersec;
    NYBRDR();
```

(continued on page 26)

Listing Three *(Listing continued, text begins on page 20.)*

```
    BOOL mOpenRead(char* _pszInFile);
    BOOL mNextSample();
    void mClose();
  protected:
    BYTE inbuf; // first nybble is high, second is low.
    BOOL nyb2;  // true if second nybble
    inline BOOL iReadNybble(BYTE& _nybble);
    };
class NYBWRT : public NYBUSR
    {
  public:
    NYBWRT();
    BOOL mOpenWrite(char* _pszOutFile, const PCMWAVEFORMAT* _pFmt);
    BOOL mWriteSample(WORD _sample, WAVEPLAYER* _pOut = NULL);
    void mClose();
  protected:
    BYTE outbuf;    // first nybble is high, second is low.
    BOOL nyb2w;
    inline BOOL iWriteNybble(BYTE _nybble);
    WORD    outsamp, intsamp;   // output and intermediate samples
    signed  diff;   // compensation, to prevent cumulative error.
    BOOL iNegFeedbackStage();
    // internal to mWriteSample (dealing with groups)...
    WORD groupbuf[SAMPLESPERGROUP];
    WORD* pgroup;
    WORD diffmin, diffmax;  // for determining magnitude.
    WORD prevsample;
    };
```

Listing Four

```
//******************************** NYBLIB.CPP *********************************
// Routines to deal with nybble format compressed waveforms.
// Copyright (c) 1993 by Neil G. Rowland, Jr. 24-JUN-93
extern "C"
    {
    #include <math.h>
    #include <stdlib.h>
    #include <string.h>
    }
#include "nyblib.h"
//*****************************************************************************
static WORD PreAdj[12+1] =  {
                    // table to map delta translating to nybble to Ø to a Ø...
    32768-8,
    32768-16, 32768-32, 32768-64, 32768-128,
    32768-256, 32768-512, 32768-1024, 32768-2048,
    32768-4096, 32768-8192, 32768-16384, Ø};
static BOOL fNagged = FALSE;
inline void Shrite(WORD& _sample, BYTE _shrite)
    { // get delta down to a nybble...
    WORD    orig = _sample;     // helps in clipping.
    if (_shrite == Ø)  { _sample = 8;  return; }
    _sample -= PreAdj[_shrite];
    _sample >>= _shrite;
    if (_sample > 15)  { // overflow/underflow.
        //D if (!fNagged)
        //D  { MessageBox((HWND)NULL, "Shrite:overflow", "NYBLIB",
        //D                                              MB_OK);  fNagged = TRUE; }
        _sample = (orig & Øx8000)? 15:0;
        }
    }
inline void UnShrite(WORD& _sample, BYTE _shrite)
    { // convert a nybble to a delta...
    if (_shrite == Ø)  { _sample = 32768;  return; }
    _sample <<= _shrite;
    _sample += PreAdj[_shrite];
    }
inline void Diff(WORD& _sample, WORD& prevsample)
    { // called once per sample.
    _sample >>=1; _sample+= 16384;  // halve magnitude to avoid overflow.
    WORD    sample = _sample;
    _sample = (sample-prevsample) + 32768;
    prevsample = sample;
    };
inline void Integ(WORD& _sample, WORD& prevsample)
    { // called once per sample.
    unsigned    intl;
    intl = prevsample;  intl +=_sample-32768;
    _sample = intl;
    prevsample = _sample;
    // undo halving and catch overflow...
    _sample -= 16384;
    _sample = (_sample&Øx8000)? (prevsample-16384)<<1 : _sample<<1;
    };
//*****************************************************************************
NYBUSR::NYBUSR()
    { hmmio = NULL;  groupcount = Ø;  encprevsample = decprevsample = 32768; }
inline void NYBUSR::iEncode(WORD& _sample)
    { // encode the data, does not include cumulative error prevention.
    if (shrite < 12)    // otherwise no gain, maybe even loss
        Diff(_sample, encprevsample);
    else
        encprevsample = _sample;    // in case last in group
    Shrite(_sample, shrite);
    if (_sample>15)  MessageBox((HWND)NULL, "Bad result", "NYBUSR", MB_OK);
    }
inline void NYBUSR::iDecode(WORD& _sample)
    { // decode the encoded data. doubles as cumulative error prevention.
    //D if (_sample > 15)
    //D { MessageBox((HWND)NULL, "bad _sample", "NYBUSR::mDecode",
    //D                                                  MB_OK); return; }
    UnShrite(_sample, shrite);
    if (shrite < 12)    // otherwise no gain, maybe even loss
        Integ(_sample, decprevsample);
    else
        decprevsample = _sample;    // in case last in group
    }
```

```
//*****************************************************************************
void NYBRDR::NYBRDR()
    { nyb2 = FALSE; }
BOOL NYBRDR::mOpenRead(char* _pszInFile)
    {
    if (!RIFFUSR::iOpenRead(_pszInFile, (char*)&Fmt, sizeof(PCMWAVEFORMAT),
                                                        'N', 'Y', 'B', '1'))
        return FALSE;
    // Make sure the input file is a mono PCM WAVE file.
    if ((Fmt.wf.wFormatTag != WAVE_FORMAT_PCM))  goto ERROR_BAD_FORMAT;
    if (Fmt.wf.nChannels != 1)  { // stereo not supported yet.
        MessageBox((HWND)NULL, "This is a STEREO wave file header",
                                        "NYBRDR", MB_ICONEXCLAMATION|MB_OK);
        goto ERROR_BAD_FORMAT1;      // bad input file format
        }
    samplespersec = Fmt.wf.nSamplesPerSec;
    groupcount = Ø;
    return TRUE;
  ERROR_BAD_FORMAT:
    MessageBox(NULL, "Input file must be a NYB1 file",
                            "NYBRDR::mOpenRead", MB_ICONEXCLAMATION| MB_OK);
  ERROR_BAD_FORMAT1:
    mClose();
    return FALSE;
    }
inline BOOL NYBRDR::iReadNybble(BYTE& _nybble)
    {
    if (nyb2)  _nybble = inbuf & Øxf;
    else  {
        if (!iReadByte(inbuf))  return FALSE;
        _nybble = inbuf >> 4;
        }
    nyb2 = !nyb2;
    return TRUE;
    }
BOOL NYBRDR::mNextSample()
    {
    BYTE    nybble;
    if (groupcount == Ø)
        if (!iReadNybble(shrite))  return FALSE;
    if (shrite)
        { if (!iReadNybble(nybble))  return FALSE; }
    else  nybble = 8;   // we don't store empty groups.
    sample = nybble;  iDecode(sample);
    groupcount++;  if (groupcount == SAMPLESPERGROUP)  groupcount = Ø;
    return TRUE;
    }
void NYBRDR::mClose()
    { RIFFUSR::iCloseRead(); }
//*****************************************************************************
void NYBWRT::NYBWRT() {
    nyb2w = FALSE; diff = Ø;
    pgroup = groupbuf;
    diffmin = diffmax = prevsample = 32768;
    }
BOOL NYBWRT::mOpenWrite(char* _pszOutFile, const PCMWAVEFORMAT* _pFmt)
    {
    if (!RIFFUSR::iOpenWrite(_pszOutFile, (const char*)_pFmt,
                                    sizeof(PCMWAVEFORMAT), 'N', 'Y', 'B', '1'))
        return FALSE;
    groupcount = Ø;
    return TRUE;
  cantwrite:
    mClose();
    return FALSE;
    }
void NYBWRT::mClose()
    { RIFFUSR::iCloseWrite(); }
inline BOOL NYBWRT::iWriteNybble(BYTE _nybble)
    { // write out to file:
    if (_nybble > 15)  { MessageBox((HWND)NULL, "Bad nybble",
                                                "NYBWRT", MB_OK);  return FALSE; }
    if (!nyb2w)  outbuf = _nybble << 4;
    else {
        outbuf |= _nybble;
        if (!iWriteByte(outbuf))  return FALSE; // to the file.
        }
    nyb2w = !nyb2w;
    return TRUE;
    }
BOOL NYBWRT::iNegFeedbackStage()
    { // encode and fight cumulative error
      // called once per sample.
    intsamp = sample;
    iEncode(intsamp);
    // fight cumulative error by applying a bias to intsamp
    if (diff < Ø)  { if (intsamp < 15)  intsamp ++; }
    if (diff > Ø)  { if (intsamp > Ø)  intsamp --; }
    if (!iWriteNybble(intsamp))  return FALSE;
    // see what the output would be, and figure compensation...
    outsamp = intsamp;
    iDecode(outsamp);
    if (outsamp > sample)  diff = 1;    // overshot
    else  if (outsamp < sample)  diff = -1; // undershot
    else  diff = Ø;
    sample = outsamp;   // in case caller is interested.
    return TRUE;
    }
BOOL NYBWRT::mWriteSample(WORD _sample, WAVEPLAYER* _pOut)
    { // takes a stream of samples divides it into groups, passes it
    WORD        diff1;
    WORD        samp16 = _sample;
    WORD        samp15 = samp16 >> 1;
    diff1 = samp16;  Diff(diff1, prevsample);
    if (diff1 > diffmax)  diffmax = diff1;
    if (diff1 < diffmin)  diffmin = diff1;
    if (++groupcount >= SAMPLESPERGROUP)  { // end-of-group processing...
        //  first determine the magnitude of the group:
        WORD diffrange = diffmax-diffmin;
        if (diffrange < ØxFØØØ)  diffrange += diffrange>>4;
        if (diffrange == Ø)  shrite = Ø;    // signifies silence
        else  { // shrite is bits delta shifted right
            shrite = 12;
            while (shrite>Ø && !(diffrange&Øx8000))
                { shrite--;  diffrange <<= 1; }
            }
        iWriteNybble(shrite);
        if (shrite)  { // process the group...
            pgroup = groupbuf;
            for (groupcount=Ø; groupcount<SAMPLESPERGROUP;
                                                    groupcount++) { // 2nd pass.
                sample = *(pgroup++);
                if (_pOut)  _pOut->mPlaySample(sample);
                if (!iNegFeedbackStage())  return FALSE;
                //D if (_pOut)  _pOut->mPlaySample(sample);
                }
            }
        // reset for next group:
        diffmin = diffmax = 32768;
        groupcount = Ø;
        pgroup = groupbuf;
        }
    *(pgroup++) = samp16;
    return TRUE;
    }
```

Listing Five

```
//******************************** NYB1.CPP ***********************************
// Main app for NYB1 compressed waveform utility.
// Copyright (c) 1993 by Neil G. Rowland, Jr. Ø4-JUN-93
extern "C" {
    #include <math.h>
    #include <stdlib.h>
    #include <string.h>
    }
#include "nyblib.h"
#define IDM_ABOUT       11          // menu items
#define ID_INPUTFILEEDIT    1Ø1     // input file name edit box
#define ID_OUTPUTFILEEDIT   1Ø2     // output file name edit box
//*****************************************************************************
int PASCAL WinMain(HANDLE hInst, HANDLE hPrev, LPSTR lpszCmdLine,int iCmdShow);
BOOL FAR PASCAL AboutDlgProc(HWND hwnd, unsigned wMsg,WORD wParam,LONG lParam);
BOOL FAR PASCAL CvtDlgProc(HWND hwnd, unsigned wMsg, WORD wParam, LONG lParam);
//*****************************************************************************
char        gszAppName[] = "Waver";   // for title bar, etc.
HANDLE      ghInst;                   // app's instance handle
void PlayIt(char* _pszNybFile)
    {
    WAVEPLAYER  Play;               // audio output.
    PCMWAVEFORMAT   pcmWaveFormat;  // contents of 'fmt' chunks
    NYBRDR      NybIn;      // for playback of nybble file
    if (!NybIn.mOpenRead(_pszNybFile))  {
        MessageBox(NULL, "Cannot read input NYB1 file", _pszNybFile,
                                                    MB_ICONEXCLAMATION | MB_OK);
        return;
        }
    Play.mOpen(&NybIn.Fmt.wf);
    for (;;)  {
        if (!NybIn.mNextSample())  break;
        Play.mPlaySample(NybIn.sample);
        }
    NybIn.mClose();
    Play.mClose();
    }
void DoIt(char* _pszInFile, char* _pszNybFile)
    {
    WAVRDR      In;
    NYBWRT      Out;
    WAVEPLAYER  Play;       // audio output.
    long        lSamples;           // number of samples to filter
    unsigned char cThis = Ø;
    signed char olddelta = Ø;
    BOOL        lastwasendpoint = TRUE;
    if (_pszInFile && *_pszInFile)  {
        // provide a default:
        if (!_pszNybFile || !lstrlen(_pszNybFile)) _pszNybFile = "\x.nyb";
        // Open the input file for reading using buffered I/O.
        if (!In.mOpenRead(_pszInFile))
            goto ERROR_CANNOT_READ;
        if (!Out.mOpenWrite(_pszNybFile, &In.Fmt))
            goto ERROR_CANNOT_WRITE;
        Play.mOpen(&In.Fmt.wf);
        for (lSamples = In.ck.cksize; lSamples > Ø; lSamples--)  {
            WORD    sample;
            if (!In.mNextSample())  goto ERROR_CANNOT_READ;
            sample = In.sample;     // raw input
            Out.mWriteSample(sample, &Play);
            }
        Out.mClose();
        Play.mClose();
        }
    // now play it back...
    PlayIt(_pszNybFile);
    goto EXIT_FUNCTION;
  ERROR_CANNOT_READ:
    MessageBox(NULL, "Cannot read input file",
                                        gszAppName, MB_ICONEXCLAMATION | MB_OK);
    goto EXIT_FUNCTION;
  ERROR_CANNOT_WRITE:
    MessageBox(NULL, "Cannot write output NYB1 file", gszAppName,
                                                    MB_ICONEXCLAMATION | MB_OK);
    goto EXIT_FUNCTION;
  EXIT_FUNCTION:
    // Close the files (unless they weren't opened successfully).
    In.mClose();
    Out.mClose();
    Play.mClose();
    }
```

```
//*****************************************************************************
int PASCAL WinMain(HANDLE hInst, HANDLE hPrev, LPSTR lpszCmdLine, int iCmdShow)
    {
    FARPROC fpfn;
    HWND    hwd;
    MSG     msg;
    // Save instance handle for dialog boxes.
    ghInst = hInst;
    if (lpszCmdLine && *lpszCmdLine)
        DoIt(NULL, lpszCmdLine);
    // Display our dialog box.
    fpfn = MakeProcInstance((FARPROC) CvtDlgProc, ghInst);
    if (!fpfn)  goto erret;
    hwd = CreateDialog(ghInst, "LOWPASSBOX", NULL, fpfn);
    if (!hwd)  goto erret;
    ShowWindow(hwd, TRUE);  UpdateWindow(hwd);
    while (GetMessage(&msg, NULL, Ø, Ø))
        if (!IsDialogMessage(hwd, &msg))
            DispatchMessage(&msg);
    DestroyWindow(hwd);
    FreeProcInstance(fpfn);
    return TRUE;
  erret:
    MessageBeep(Ø);
    return FALSE;
    }
// AboutDlgProc - Dialog procedure function for ABOUTBOX dialog box.
BOOL FAR PASCAL AboutDlgProc(HWND hWnd, unsigned wMsg,WORD wParam,LONG lParam)
    {
    switch (wMsg)  {
    case WM_INITDIALOG:
        return TRUE;
    case WM_COMMAND:
        if (wParam == IDOK)
            EndDialog(hWnd, TRUE);
        break;
    }
    return FALSE;
    }
// CvtDlgProc - Dialog procedure function for conversion dialog box.
BOOL FAR PASCAL CvtDlgProc(HWND hWnd, unsigned wMsg, WORD wParam, LONG lParam)
    {
    FARPROC     fpfn;
    HMENU       hmenuSystem;    // system menu
    HCURSOR     ghcurSave;      // previous cursor
    switch (wMsg)  {
    case WM_INITDIALOG:
        // Append "About" menu item to system menu.
        hmenuSystem = GetSystemMenu(hWnd, FALSE);
        AppendMenu(hmenuSystem, MF_SEPARATOR, Ø, NULL);
        AppendMenu(hmenuSystem, MF_STRING, IDM_ABOUT,
            "&About LowPass...");
        return TRUE;
    case WM_SYSCOMMAND:
        switch (wParam)  {
        case IDM_ABOUT:
            // Display "About" dialog box.
            fpfn = MakeProcInstance((FARPROC) CvtDlgProc, ghInst);
            DialogBox(ghInst, "ABOUTBOX", hWnd, fpfn);
            FreeProcInstance(fpfn);
            break;
        }
        break;
    case WM_COMMAND:
        switch (wParam)  {
        case IDOK:          // "Begin"
            // Set "busy" cursor, filter input file, restore cursor.
            char        szInFile[200];      // name of input file
            char        szOutFile[200];     // name of output file
            // Read filenames from dialog box fields.
            szInFile[Ø] == Ø;
            GetDlgItemText(hWnd, ID_INPUTFILEEDIT, szInFile,
                                                    sizeof(szInFile));
            szOutFile[Ø] == Ø;
            GetDlgItemText(hWnd, ID_OUTPUTFILEEDIT, szOutFile,
                                                    sizeof(szOutFile));
            ghcurSave = SetCursor(LoadCursor(NULL, IDC_WAIT));
            DoIt(szInFile, szOutFile);
            SetCursor(ghcurSave);
            break;
        case IDCANCEL:      // "Done"
            PostQuitMessage(Ø);
            break;
        }
        break;
    }
    return FALSE;
}
```

End Listings

Multimedia Audio Systems

Will General MIDI lead the way in interactive music?

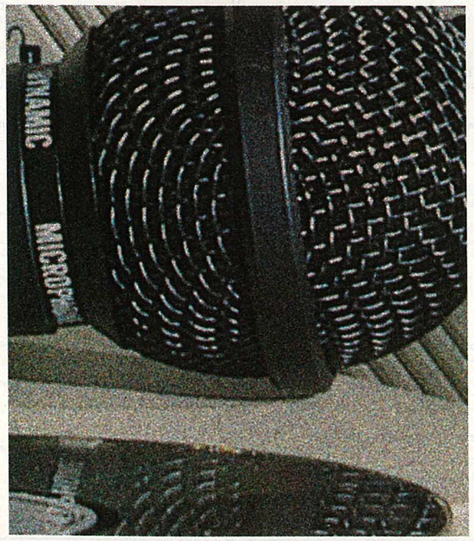
John Ratcliff

If any single segment of the multimedia-hardware market has boomed over the last two years, it's sound. Spurred on by the success of Creative Labs' SoundBlaster and Windows 3.1, most vendors (hardware and software) are supporting multimedia sound in one way or another. From Turtle Beach's Multisound and Roland's Sound Canvas at the high-end, to the SoundBlaster and MediaVision's Thunderboard at the low-end, users can take their pick, depending on price, features, and performance.

What does all of this activity mean? For the user it means chaos and confusion; for the hardware vendors, bitter rivalries and vicious competition. Still, at the center of this is a fascination with producing music and sound effects inside software, a captivation that goes beyond that of even hardcore audiophiles. People are passionate about music, and thus the intense interest in this burgeoning technology.

The sound industry is currently in a great state of flux. While most of the two million or so installed sound cards use reliable, albeit basic, digital sound and the familiar FM (kazoo-like) synthesis, the current debate involves the type of audio device yet to come. What are the features of the next-generation sound system? Who is going to provide these systems, in what package, with what features, and at what price point? And the $64,000 question is, "Who needs next-generation sound and why?"

As a longtime observer of and participant in this industry, I'll address some of the questions about where we're heading. As for the "why" we're heading in this direction, I'm not entirely clear. Multimedia technology is, to a great extent, a solution without a problem. It's a collection of gee-whiz toys that are fun, but not yet particularly useful. Certainly many aspects of multimedia greatly enhance the enjoyment and appeal of applications software. Maybe that's enough.



What most people accept as a definition of multimedia is what computer-game developers have been trying to provide since *Space War* first popped up on a university mainframe. Computer games are pushing the envelope and defining the nature of how sound and music are used in an interactive environment. This is evidenced by products such as Virgin/Trilobyte's *The 7th Guest.* This interactive film on CD-ROM has an original score by the Fat Man (aka George Sanger, an Austin, Texas-based composer of music for computer games like *The 7th Guest* and *Wing Commander*; for more on the Fat Man, see my article, "Examining PC Audio," *Dr. Dobb's Journal,* March 1993) and a fully orchestrated "Red Book," or CD-DA, audio sound track featuring vocals and music by three Grammy winners. Another example is LucasArts' *X-Wing,* which uses the John Williams score from the *Star Wars* film as arranged for general MIDI and adapted for interactivity by Michael Land, Peter McConnel, and Clint Bajakian. In addition to The Fat Man, other interactive music composers include Rob Wallace of Wallace Music and Sound (Glendale, Arizona) Donald Griffin of Computer Music Consulting (San Francisco, California) and Bobby Prince (Venice, Florida).

The way in which sound and music are used in an interactive environment is critical to the success of multimedia. As the Fat Man said in the March 1993 issue of *Dr. Dobb's Journal:*

> Sound and music can increase throughput, enhance the pleasantness of computer experience, and increase entertainment value of a program.
>
> When should music and sound be used? Not just in games.
>
> Although computers and film are very different media, the relative maturity of the latter makes it useful for developers to look at film as a model for the future of some aspects of computer software. Especially with the importance of multimedia, we can look for examples not only in feature films, but all kinds of video applications: educational and industrial films, training, advertising, and news programs—anything that's been on a film or videotape.
>
> When do they use music and sound? To enhance emotion where it already exists. Some folks like happy faces or other graphics when their computers start up—a happy tune can triple the happy effect.
>
> Sound also manipulates or changes an emotion that might already exist. A child safety multimedia presentation might show a picture of a cute baby near a swimming pool, something like the *Jaws* theme might keep the user from focusing on the cuteness of the baby. A short, simple title tune might make a database program seem less complex and frightening. In the case of the "bad news" dialog box, consider how much more palatable a warning a pleasant "ping" is than an explosive sound (or a picture of a bomb, for that matter.) With audio, the developer, like a film director is able to control the degree of emotion the user feels.

The general MIDI specification uses wave-table synthesis and incorporates digitally sampled instruments to produce rich symphonic sound

How should music and sound be used in computer programs? Like graphics, they have to be done right. Simply filling up space with spectacular graphics done by an "artist friend" (everybody's got one) is simply not the way to create an effective program, and the same applies to music. There's as much an art to using music and sound in software as there is in film. And of course it's too big a subject to address here.

Regardless of the extent to which music and sound will be used in multimedia applications, it's the computer-game industry that's on the leading edge, helping define standards and pushing the limits of the technology. I've worked with more than a hundred game publishers to incorporate music and sound in over 200 commercial products. From this vantage point, it seems that "general MIDI" is the best way to deal with interactive music in applications software.

The Musical Instrument Digital Interface (or "MIDI") specification is an internationally supported de facto standard that defines a serial interface for connections between music synthesizers, musical instruments, and computers. MIDI, which is maintained by the MIDI Manufacturers Association (Los Angeles, California), is based both on hardware (I/O channels, cables, and the like) and software (encoded messages defining device, pitch, volume, and so forth). According to the specification, the receiving device in a MIDI system interprets the musical data even

MANUFACTURER:	Sound Man 16 Logitech	Sound Man Games Logitech	Multisound Turtle Beach	Pro Audio Spectrum 16 (Macintosh) Media Vision	Pro Audio Studio Media Vision	Pro Audio BASIC Media Vision	Pro Audio Spectrum 16 Media Vision	Pro 16 Multimedia System II Media Vision	Fusion Double CD 16E or 16 Media Vision	Fusion CD 16E or 16 Media Vision	SC-7PC Roland	RAP-10/AT Roland	SCC-1B Roland
List price	$199	$139	$599	$395/495	$349	$199	$299	$1195	$999/899	$699	$399	$599	$499
Installed base	NP	NP	NP	NP	NP	NP	NP	NP	NP	NP	NP	NP	NP
Bus Interface (1)	ISA	ISA	ISA	LC/NuBus	ISA	ISA	ISA	ISA	ISA	ISA	ISA	ISA	ISA
DIGITAL SECTION													
No. of channels	1	1	2	2	3	3	3	3	3	3	NA	2	NA
Stereo	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	NP	yes	NP
Resolution	16	8	8/16	8/16	8/16	8/16	8/16	8/16	8/16	8/16	NP	8/16	NP
DMA	yes	yes	no	NA	yes	yes	yes	yes	yes	yes	NP	yes	NP
Compression	yes	yes	yes	no	no	no	no	no	no	no	NP	no	NP
SoundBlaster compatibility	yes	yes	no	NA	yes	yes	yes	yes	yes	yes	NP	no	NP
FM SECTION													
No. of voices	20	20	32	20	20	20	20	20	20	20	26/16	26/16	26/16
Stereo	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
Adlib compatibility	yes	yes	no	NA	yes	yes	yes	yes	yes	yes	no	no	no
OPL3 chip set (2)	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	no	no	no
MIDI SECTION													
On-board MIDI controller	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
General MIDI patch set	no	no	yes	NP	NP	NP	NP	NP	NP	yes	yes	yes	yes
DIGITAL API SUPPORT													
Windows Wave driver	yes	yes	yes	NA	yes	yes	yes	yes	yes	yes	NP	yes	NP
Linkway driver	NP	no	no	NA	yes	yes	yes	yes	yes	yes	NP	yes	yes
MIDI API SUPPORT													
Windows MIDI driver	yes	yes	yes	NA	yes	yes	yes	yes	yes	yes	yes	yes	yes
Programmer's kit	no	no	yes	NP	yes	yes	yes	yes	yes	yes	yes	yes	yes

(1) ISA, Standard PC bus interface; MCA, Microchannel; (2) supports Yamaha GSS standard, with OPL3 chipset; provides 20 channels or four operator FM voices.
NA=not applicable, NP= not provided.

(continued on page 30)

John is a graphic artist, designer, and programmer living in St. Louis. His entertainment products include 688 Attack Sub *and* Seawolf *from Electronic Arts and KaleidoSonics from Masque publishing. He can be contacted at 747 Napa Lane, St. Charles, MO 63303.*

though the sending device has no way of knowing what the receiver can do. But this can be a problem if the receiving device doesn't have the capability to interpret the data correctly. General MIDI addresses this problem by identifying hardware capabilities in advance.

All general-MIDI devices have 128 sound effects as well as musical instrument and percussion sounds. General-MIDI systems support simultaneous use of 16 MIDI channels with a minimum of 24 notes each, and they have a specified set of musical controllers. This means that with general MIDI, the sender knows what to expect of the receiver. Consequently, a file created with one general-MIDI device is recognizable when played on any other—without losing notes or changing instrumental balance.

General MIDI, a platform-independent subset of MIDI that's being supported by a wide-range of hardware and software vendors, has been endorsed by more than 100 companies, including Microsoft, Apple, IBM, WordPerfect, and the like. The specification is supported on Windows, DOS, Macintosh, Atari, and Amiga. In fact, general MIDI is the authoring format for MIDI required by Windows and MPC. Currently, the MIDI Manufacturers Association is defining general-MIDI file-format standards.

Multimedia is a solution without a problem—a collection of gee whiz toys, that while fun, aren't particularly useful

Sound boards that provide general-MIDI support include the MediaVision's Pro AudioStudio 16 and Creative Labs' SoundBlaster 16 with general MIDI wave-sample synthesis daughterboard. Furthermore, general-MIDI chipsets are being developed by a number of companies. I'll touch on some of these boards and chipsets later in this article.

From a practical viewpoint, general MIDI is widely acknowledged as being defined by Roland Corporation's SoundCanvas (SCC55 and SCC1). The Roland SoundCanvas is a wave-table synthesis MIDI device that supports the general-MIDI specification and implements chorus and reverb effects that give the music a rich timbre. (Unlike FM synthesis, which begins with a generic sound wave, wave-table synthesis uses recordings of actual instruments to generate music.) The result is that when a user plays *X-Wing* with a Roland SoundCanvas, and the familiar John Williams score kicks in, the music is comparable to the actual film score. The Roland SoundCanvas and the general-MIDI specification both support rich symphonic sound. Using wave-table synthesis that incorporates extremely high-resolution, digitally sampled instruments gives you rich string sections, unbelievable piano, and percussion that rocks your socks off.

Another recent development that's bridged the conversion to general MIDI is Yamaha's license of the Fat Man's general-MIDI patch set for the OPL2 and OPL3 FM synthesizers—the same FM synthesizers found in sound cards from Adlib, Creative Labs, MediaVision, and Microsoft. The Fat Man developed a set of tones that allow fully orchestrated general-MIDI music to be played on sound cards such as the SoundBlaster. These tones, available to all developers, are being incorporated into computer games and Windows drivers. Though FM synthesis chips like the OPL2 or OPL3 can never play music that sounds like a wave-table device, their tones convey a general-MIDI score with a high level of musicality.

While it's beyond the scope of this article to go into any detail about the myriad of general-MIDI sound cards crowding the multimedia market, I'll nonetheless briefly mention each. Table 1 provides further details. The cream of the crop is the Roland SoundCanvas (SC55 and SCC1). Roland is now shipping the RAP-10, a lower-cost version with digital sound support. Turtle Beach has one of the most advanced multimedia sound cards, Multisound, which not only supports general MIDI, but features a full Proteus 1XR and the highest quality digital sound support. Turtle Beach has a new low-cost general-MIDI synthesizer in development called the "Maui" card. Creative Labs' entry into the general-MIDI market is their WaveBlaster card, a daughterboard for the already-popular SoundBlaster 16.

Sierra Semiconductor has developed a general-MIDI hardware specification called "Aria" which provides general MIDI, digital audio, and SoundBlaster compatibility in a low-cost package. Sierra has also licensed Archer Communications' QSound Virtual Audio technology for integration into the DSP-based Aria chipsets. QSound Virtual Audio is a multidimensional sound-localization technology that allows sounds to appear to be coming from locations unreferenced to the speakers. Programmers can use QSound to generate "soundscapes" that exceed the physical bounds of stereo-speaker geometry. The Aria chipset consists of a controller, DSP, and memory. It has been adopted by a number of vendors and is used in MidiMaestro from Computer Peripheral and SonicSound from Diamond Computer Systems.

General MIDI is widely acknowledged as being defined by Roland Corporation's SoundCanvas

Ensoniq's entry into the general-MIDI market is their SQ1000 card, which also features SoundBlaster-compatibility. The Media FX sound card from Video Seven, also based on Ensoniq's Soundscape chipset, provides 32 voices in four Mbytes of ROM. It records and plays back at 44.1 kHz. Advanced Gravis has added general-MIDI support to their popular Gravis Ultrasound card. Yamaha is developing a chipset called the "OPL4" that's likely to be adopted by a number of vendors soon. The OPL4 features full general-MIDI support, as well as backward compatibility with OPL2 and OPL3 FM synthesis. Yamaha has also released its own sound card, the Hello!Music!, that's based on the chipset. At the heart of the board, which is fully MIDI compatible, is the CBX-T3 Tone Generator that provides 192 instrument sounds, 10 drum kits, and digital reverb. The CBX-T3 can also accept audio input from audio devices such as tape/radio players, CD players, or microphones.

	Multimedia Video Conferencing	SonicSound	SonicSound LC	Sounding Board	SoundWave FX	Sound Wizard Pro	VGA Stereo-F/X	Stereo-FX	Stereo F/X-CD	CD Sound Dimension	Echo DSP	Lifesize Sound Enhancer
MANUFACTURER:	ASC	Diamond	Diamond	Artisoft	Reveal	Criterion	ATI	ATI	ATI	ATI	Echo Speech	Activision
List price	$4600	$299	$199	$99/199	new	$299	$199	$129	$199	$399	$189.95	$9.95
Installed base	100	new	new	NP	new	NP	NP	NP	NP	NP	NP	NP
Bus Interface (1)	PC/AT	ISA	ISA	ISA/MCA	ISA	ISA	ISA/EISA	ISA/EISA	EISA/ISA	EISA/ISA	ISA	printer parallel port
DIGITAL SECTION												
No. of channels	8	16	16	2	2	2	2	2	2	2	2	1
Stereo	yes	yes	yes	NP	yes	yes	yes	yes	yes	yes	yes	no
Resolution	12	8/16	8/16	NP	16	16	8	8	8	8	16	8
DMA	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no
Compression	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no
SoundBlaster compatibility	optional	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no
FM SECTION												
No. of voices	2/4	32	20/24	2	32	32	11	11	20	20	9	NA
Stereo	yes	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	NP
Adlib compatibility	yes	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	NP
OPL3 chip set (2)	NP	no	yes	no	NA	NA	no	no	yes	yes	NP	NP
MIDI SECTION												
On-board MIDI controller	yes	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	no
General MIDI patch set	yes	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	no
DIGITAL API SUPPORT												
Windows Wave driver	optional	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
Linkway driver	NP	no	no	NP	yes	yes	NP	NP	NP	NP	no	NP
MIDI API SUPPORT												
Windows MIDI driver	yes	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	no
Programmer's kit	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	NP	yes

(1) ISA, Standard PC bus interface; MCA, Microchannel; (2) supports Yamaha GSS standard, with OPL3 chipset; provides 20 channels or four operator FM voices.
NA=not applicable, NP= not provided.

	Soundmaster II	Sound Blaster 16 SCSI-2	Wave Blaster	Sound Blaster 16 Basic	Sound Blaster 16	Sound Blaster 16 MultiCD	UltraSound	UltraSound MAX	PORT-ABLE Sound Plus	The Sound Source	Ad Lib Gold 1000	VGA Maestro 16	Soniq 16
	Covox	Creative Labs	Creative Labs	Creative Labs	Creative Labs	Creative Labs	Advanced Gravis	Advanced Gravis	DSP Solutions	Walt Disney	Ad Lib MultiMedia	Computer Peripherals	Best Data
List price	$229.95	$329.95	$249.95	$199.95	$249.95	$249.95	$199	$299	$198.95	$49.95	$299.95	$299	$250
Installed base	NP	new	NP	new	NP	new	100,000	100,000	8500	NP	NP	NP	NP
Bus Interface (1)	NP	ISA	NP	ISA	ISA	ISA	ISA	ISA	parallel port	parallel-port adapter	ISA	ISA	ISA
DIGITAL SECTION													
No. of channels	NP	NA	32	NA	NA	NA	32	32	2	1	2	16	2
Stereo	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	yes	yes
Resolution	8	8/16	16	8/16	8/16	8/16	16	16	8/16	NP	8/12	16	16
DMA	yes	yes	NA	yes	yes	yes	yes	yes	no	NP	yes	yes	yes
Compression	yes	yes	NA	no	no	no	no	yes	yes	NP	yes	yes	yes
SoundBlaster compatibility	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	no	yes	yes
FM SECTION													
No. of voices	NP	20	NP	20	20	20	32	32	9	NA	20	20	32
Stereo	no	yes	yes	yes	yes	yes	yes	yes	no	no	yes	yes	yes
Adlib compatibility	yes	yes	no	yes	yes	yes	yes	yes	yes	no	yes	yes	yes
OPL3 chip set (2)	no	yes	no	yes	yes	yes	yes	yes	no	no	yes	no	NA
MIDI SECTION													
On-board MIDI controller	yes	yes	no	yes	yes	yes	yes	yes	no	no	yes	yes	yes
General MIDI patch set	NP	yes	yes	yes	yes	yes	yes	yes	no	no	yes	yes	yes
DIGITAL API SUPPORT													
Windows Wave driver	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
Linkway driver	no	NP	NP	NP	NP	NP	NP	NP	yes	no	no	NP	yes
MIDI API SUPPORT													
Windows MIDI driver	yes	yes	no	yes	yes	yes	yes	yes	yes	no	yes	yes	yes
Programmer's kit	yes	no	no	no	no	no	yes	yes	yes	yes	yes	yes	yes

Logitech's SoundMan 16, which uses MediaVision's Spectrum chipset, is a latecomer to the 16-bit sound-board arena. The SoundBlaster-compatible card uses the Yamaha OPL-3 synthesizer chip to provide 20-voice MIDI support. As you can see, there are a *lot* of vendors fighting over a market that has yet to even be defined.

How do you provide general-MIDI support on such a wide array of hardware platforms? Under Windows, the multimedia interface will do a lot of the work, but it's an incomplete solution. With Windows, you can ask to play a MIDI file from disk, with no interactive control and the accompanying disk thrashing, or you can feed it MIDI by hand. In other words, there's a high-level and low-level interface, but no medium-level interface. And that's a problem.

Under DOS there's currently only one complete MIDI solution—the Audio Interface Library from Miles Design. This is a set of medium-level interface libraries for DOS real mode, DOS protected mode, and Windows. These drivers are currently used by about 70 percent of the DOS-based game-development community. Miles Design licenses these MIDI drivers to professional developers and publishers with full source code disclosure and unlimited distribution rights with a license fee. The VESA committee is currently working on a standardized interface specification for wave audio and MIDI music. (For more information, see "The VESA BIOS Extension/Audio Interface" on page 58 of this issue.) Though VESA's efforts are noble, it will be sometime before all of the various vendors agree to and implement the new standard. Once this is achieved, things will definitely improve. For the moment, however, there appear to be just two de facto standards: SoundBlaster compatibility (available through a license from Creative Labs) and general MIDI utilizing the MPU401 UART in "dumb" mode (à la Roland MT32, LAPC, SCC55, RAP-10, and so on).

Adding music and sound to your application software can be both fun and rewarding. It can also be the most frustrating. To make it more rewarding, here's my advice: Make the investment in general MIDI and find a good professional source for sound and music, either from an experienced interactive-media composer, or from quality off-the-shelf music and sound clips.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

For More Information

Activision
P.O. Box 67001
11440 San Vincent Blvd., #310
Los Angeles, CA 90049
310-207-4500

Adlib Corp.
20020 Grande Allee East, #850
Quebec City, PQ
Canada G1R 2J1
418-529-9676

Advanced Gravis
101-3750 N. Fraser Way
Burnaby, BC
Canada V5J 5E9
604-431-5020

Advanced Strategis Corp.
60 Cutter Mill Road, #502
Great Neck, NY 11021
516-482-0088

Artisoft Inc.
691 East River Rd.
Tucson, AZ 85704
800-846-9726

ASC Computer Systems
P.O. Box 566
26401 Harper Ave.
St. Clair Shores, MI 48080
313-882-1133

ATI Technologies Inc.
33 Commerce Valley Dr. East
Thornhill, ON
Canada L3T 7N6
416-882-2600

Covox Inc.
675 Conger St.
Eugene, OR 97402
503-342-1271

Creative Labs Inc.
1901 McCarthy Blvd.
Milpitas, CA 95035
408-426-6600

Computer Peripherals Inc.
67 Rancho Conejo
Newbury Park, CA 91320
800-854-7600

Diamond Computer Systems
1130 E. Arques Ave.
Sunnyvale, CA 94086
408-736-2000

DSP Solutions
2464 Embarcadero Way
Palo Alto, CA 94303
415-494-8086

Ensoniq
155 Great Valley Parkway
Malvern, PA 19355
215-647-3930

The Fat Man
7611 Shoal Creek Blvd.
Austin, TX 78757
512-454-5775

Logitech
6505 Kaiser Dr.
Fremont, CA 94555
510-795-8500

MediaVision
47300 Bayside Parkway
Fremont, CA 94538
510-770-8600

Miles Design Inc.
10926 Jollyville, #308
Austin, TX 78759
512-345-2642

Bobby Prince
P.O. Box 1436
Venice, FL 34284
813-484-4969

Roland Corp.
7200 Dominion Circle
Los Angeles, CA 90040-3647
213-685-5141

Computer Music Consulting
Donald S. Griffin
239 Richland Avenue
San Francisco, CA 94110
415-285-3852

Sequoia Systems Inc.
400 Nickarson Rd.
Marlboro, MA 01752
800-562-4593

Echo Speech Corp.
6460 Via Real
Carpinteria, CA 93013
805-684-4593

The Audio Solution
P.O. Box 11688
Clayton, MO 63105
314-567-0267

Turtle Beach Systems
52 Grumbacher Road, #6
York, PA 17402
717-767-0200

Voyetra Technologies
333 Fifth Ave.
Pelham, NY 10803
914-738-4500

Wallace Music & Sound Inc.
Rob Wallace, Executive Producer
6210 West Pershing Avenue
Glendale, Arizona 85304-1141
602-979-6201

Walt Disney Software
P.O. Box 290
Buffalo, NY 14207-0290
818-841-3326

Yamaha Corp. of America
Consumer Products Division
P.O. Box 6600
Buena Park, CA 90622-6600
714-522-9240.

NEW

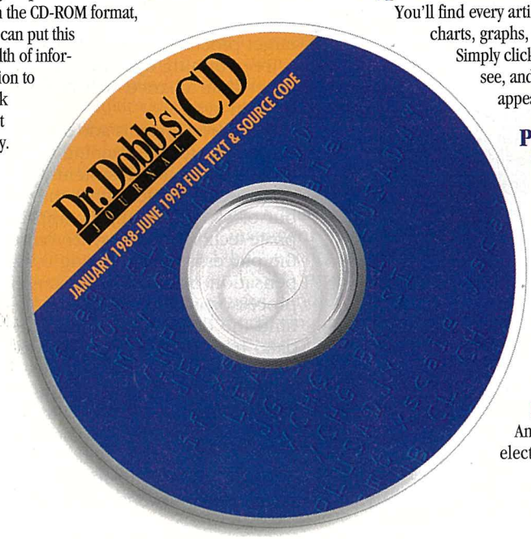
Dr. Dobb's/CD

Hundreds of Articles. Thousands of Lines of Source Code. All Accessible in Seconds.

***Dr. Dobb's Journal*, the leading authority for professional software developers, is now available on CD-ROM!** This definitive source of articles and source code allows you to quickly search for that elusive algorithm, gather valuable programming ideas, incorporate *DDJ* code directly into your programming projects, and much, much more.

Tons of Information—Including Bonus Source Code

With ***Dr. Dobb's/CD***, you get all our issues from January 1988 through June 1993, including hundreds of great articles by the best authors in the industry. Along with all tables, charts and graphs, source code, and special issues. *Plus thousands of lines of previously unpublished source code.* With the CD-ROM format, you can put this wealth of information to work right away.



Powerful, Intuitive Search Capabilities

A wide range of search options makes locating information easy and fast, requiring just seconds. For example, you can search the Table of Contents on progressive levels: by year, issue, title, subject, author, article, or source-code listings. Or conduct a full-text search, with Boolean search capabilities. Search for ranges, partial words, do proximity searches, use wildcards. Expand or narrow your searches. Store your "hits." Document your search paths. Uncover related material you otherwise might have missed. Your CD-ROM harnesses the most powerful search engine available today.

Hypertext Links

You'll find every article hotlinked to its tables, charts, graphs, and source-code listings. Simply click on the item you need to see, and the information will appear instantly.

Pull Information, Then Use It

Using the Windows version, copy and paste full or selected portions of articles to the clipboard to later paste into your programming project. DOS users can copy text or code directly into any file.

Add Your Own Notes

Annotate your searches with electronic notes, then save them to your hard drive. Handy bookmarks let you quickly find your place anywhere on the CD you leave off.

On-Line Help

Context-sensitive help is always available to guide you through your searches.

Legendary First Issue

Dr. Dobb's/CD comes with an added feature—an electronic version of the first issue of *Dr. Dobb's Journal*, published in 1976. See it exactly as it was printed back in the days when BASIC was king—and *DDJ*'s Tiny BASIC was the crown prince!

Platforms Available: DOS and WINDOWS

Valuable programming tools are yours on a silver platter. Buy *Dr. Dobb's/CD* today!

Introductory Price: $79.95
(plus $2.00 shipping)

Free Call!
800-759-6684

Call 24 hours a day, 7 days a week. Visa/MasterCard accepted. Outside the U.S. & Canada please use fax (404) 426-1044, or mail orders to: ***Dr. Dobb's/CD,*** P.O. Box 105843, Atlanta, GA 30348-9523, USA.

30 Day Money Back Guarantee

Dr. Dobb's/CD, Miller Freeman, Inc., 411 Borel Avenue, Ste. 100, San Mateo, CA 94402, 415-358-9500

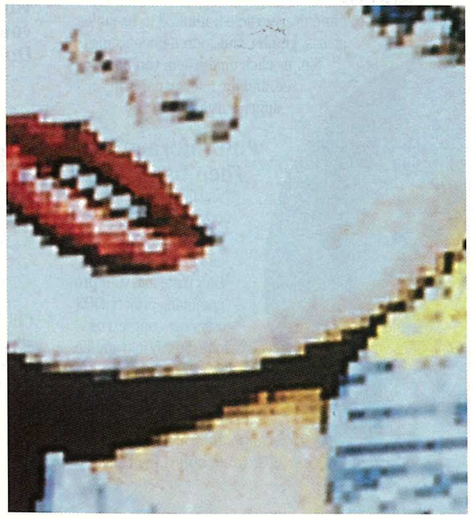
Inside OS/2 Software Motion Video

Using threads to synchronize audio and video data

Les Wilson

Until recently, digital video's huge demands for processing power and data storage were major hurdles for PC developers. To a great extent, recent advances in hardware, CD-ROM, and image-compression technologies have enabled us to make gains in the race for realistic digital video. Fully synchronizing audio and visual data, however, is one of the technical challenges yet to be solved. You know the problem: "Out-of-sync" audio and video in foreign monster films where English-speaking voices are dubbed over a non-English-speaking actor's moving lips. For low-budget entertainment, we've been generally tolerant of this lack of synchronization. However, system providers must address the timing and synchronization problems to ensure the serious use and acceptance of their system.

IBM's Multimedia Presentation Manager/2, Apple's QuickTime, and Microsoft's Video for Windows all provide users with the ability to create and manipulate digital video and audio data. To do software motion-video playback, such a system must first locate the data for presentation and transport it from its current location to the playback system. Depending on the way the data was created, the task of locating and transporting the data can be simple or complex. In the simple case, the system opens the file and reads a buffer of data. In the complex case, the system traverses data structures and retrieves the required data using indirect pointers to other files.



Next, the system segregates and moves each type of data to the appropriate processor for that data. Audio data goes to the audio subsystem, video goes to the software decompressor, and so on. Synchronization of the data presentation occurs at the target end of the pipeline; see Figure 1.

Types of Synchronization

There are two types of synchronization: *free-running* and *monitored*. The free-running technique cues up the audio and video data at the target processor, kicks them off, crosses its fingers, and hopes for the best. Sometimes it works, sometimes it doesn't. Systems that use this technique alone often exhibit inconsistent audio and video sync. This is especially common when processing high-motion scenes in the video that cause the target video processor to lag behind the audio. Other interference from device contention can also cause a given target processor to lose synchronization. Once a target processor is out of sync in a free-running system, nothing other than chance will bring it back in sync.

Monitored systems add a policing activity to free-running target processors. In these systems, target processors detect when they're out of sync and employ appropriate techniques to resynchronize the processed data. Timing compensation occurs when a target processor is either ahead or behind the desired location in the data. In doing so, target processors adjust their processing speed depending on the complexity of the data, processing power available on the system, and interference from outside activities.

Monitored systems react to the complex interactions occurring in the system. For audio/visual data, this type of system constantly resynchronizes what is seen with what is heard. This doesn't help the foreign actor with the wrong lip movements for the English-language sound track, but it does ensure that the "thud" is heard when the monster hits the ground.

Using Interleaved Data

Both free-running and monitored systems are affected by how the data is organized at the source. When audio and video data is evenly distributed, or *interleaved*, it flows easily into the system with minimal overhead. That is, as long as there's sufficient data for the target processors, the source data can be read in a single sequential stream. Free-running systems work best when audio and video data are interleaved. Interleaving is also very good when data is on slow devices such as CD-ROM.

While interleaving can help synchronization, it shouldn't be mandatory. Some file formats allow audio and video data to be "clumped" at the beginning or the end of a file. Other file formats allow the data to be distributed in other local and remote files. Either way, multimedia systems must ensure adequate processing time is allocated to prefetch the required data to affect on-time delivery to the target processors.

Using Multiple Threads for Synchronization

The OS/2 data-streaming model employs the concept of source and target stream handlers. Chains of stream handlers process and move data at discrete points in a stream of data. As the stream handler moves data from point to point, it performs the required data-specific timing and processing operations. In this way, the stream handler becomes a convenient place to encapsulate data-dependent and timing-dependent operations.

OS/2's Multimedia Presentation Manager implements this streaming model with several independently dispatched threads. Each stream handler is a thread that controls the processing of data through a certain point in the stream. Additionally, a centralized buffer-management and timing thread called the "synchronization stream manager" (SSM) provides services to each stream handler for handling buffers and monitoring its own processing against that of other stream handlers. Together, these threads join to move, filter, and present data with monitored synchronization. This is illustrated in the data and control flows diagramed in Figure 2.

Here's what happens during playback of a software motion-video file:

1. The application initiates the playback operation using the media-control interface (MCI) API. These operations identify the source file and the operation(s) to be performed (play, pause, seek, and so on).
2. The system loads the movie and starts the play.
3. The digital-video media-control driver (MCD) uses the Multimedia I/O services (MMIO) to find and open the file. The MCD is isolated from the file being local or remote.
4. MMIO identifies the file format and uses one of its pluggable I/O procedures (IOPROC) to handle any file-format dependent operations. This allows support of additional file formats without modification of the MCD.
5. The IOPROC opens the file, identifies it, examines the contents, and determines the type of video it contains.
6. The IOPROC loads and initializes the appropriate software decompressor.
7. The MCD initiates the required stream handlers, and allocates the buffer management, timing services, and hardware resources required by the contents of the file.
8. When ready, the multitrack stream handler (MTSH) reads the audio and video data from the file.

Les is a senior programmer in IBM's OS/2 Multimedia Software group. He was the project leader of the team that invented and developed the software motion-video support in OS/2 2.1. The synchronization algorithm described here was invented by Steve Hancock and Bill Lawton. Les can be reached at IBM Corp., 1000 NW 51st St., Boca Raton, FL 33431.

32 BIT Graphics Power!

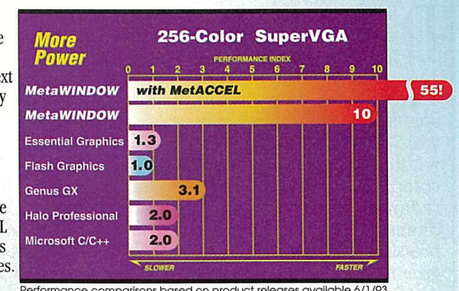
MetaWINDOW
Power Graphics Programming!

Text and Printer support built-in. 3D Motif-style GUI C source code included free!

NEW! – MetaWINDOW-DOS/386
Advanced Graphics for 32-Bit Protected-Mode
All Products Royalty-Free!

Add advanced graphics power to your application programs using MetaWINDOW! Optimized in tight assembly language code, MetaWINDOW provides an extensive set of graphic functions, top-rated drawing speed, and direct support for virtually every graphics card and SuperVGA available.

MetaWINDOW includes advanced features for: region clipping, image scaling, object detection, multi-colored patterns, scalable vector text and more. MetaWINDOW also fully supports drawing to off-screen bitmaps located in conventional, EMS, XMS or virtual disk memory.



Performance comparisons based on product releases available 6/1/93. Full benchmark report downloadable from Metagraphics BBS, 408/438-5368.

MetACCEL-DOS/286/386
Add support for the latest hardware accelerated SuperVGA's. MetACCEL boosts drawing speed 5 to 12 times over software-only SuperVGA modes.

MetHICOLOR-DOS/286/386
Add support for near photo-realistic imaging with MetHICOLOR. MetHICOLOR supports new 15 and 16 bit-per-pixel display modes with up to 65,536 displayable colors.

FontWINDOW/PLUS
Provides over 250 additional fonts for use with MetaWINDOW. Also includes font editor, plus FontBuilder and TypeImporter utilities for creating fonts of any size from *PostScript* or *TrueType* typefaces.

Find out today why more commercial applications use MetaWINDOW then any other graphics toolkit. Full source code is also available, plus best of all – *MetaWINDOW is royalty-free!*

MetaWINDOW – Power Graphics For Your PC!

MetaWINDOW-DOS	All Products Royalty-Free!	$249
MetaWINDOW-DOS/286		$349
MetaWINDOW-DOS/386		$449
MetACCEL-DOS/286/386		$349
MetHICOLOR-DOS/286/386		$349
PCXLab-DOS/286/386		$349
FontWINDOW/Plus		$249

30 DAY GUARANTEE MONEY BACK

Order Today 800-332-1550

METAGRAPHICS SOFTWARE CORPORATION
269 Mount Hermon Road, Scotts Valley, CA 95066
Phone: 408-438-1550 Fax: 408-438-5379

CIRCLE NO. 156 ON READER SERVICE CARD

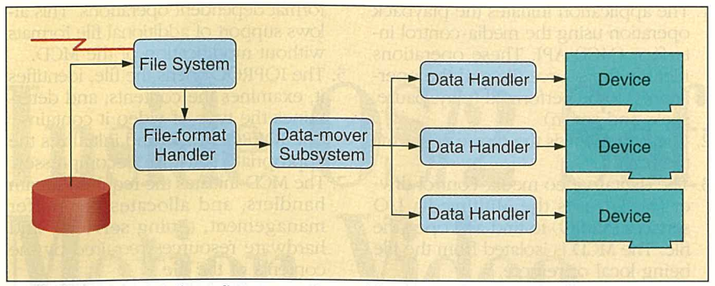


Figure 1: *Typical data flow in multimedia systems.*

9. The MTSH identifies and splits the data into its output streams.
10. The system cues the streams.
11. The MCD starts target stream handlers and controls the playback as requested by the application.
12. As the video stream handler runs, it tells the pluggable video decompressor to reconstruct and display the frame.
13. On systems that allow direct access to the display (such as OS/2), the video decompressor reconstructs the image directly into its window. Otherwise, the image is reconstructed in system memory and displayed by the video stream handler.

As the system runs, it dispatches threads according to priorities and scheduling algorithms. Each point in the stream performs its part of the entire task to move data from the source to the target. As each stream handler does its work, it records its progress so that the SSM can monitor the data stream and provide synchronization services.

Depending on a hardware platform's display hardware, the time required to display a frame can vary greatly. The less efficient the display subsystem, the less processing power is available for other activities in the system. To solve the problems of inefficient display subsystems, OS/2 2.1 allows its video decompressors to bypass the graphics subsystem and access the display adapter directly. When implemented by the adapter's OS/2 display driver, this bypass dramatically improves the performance of software motion video. (See the accompanying text box "About IBM's Ultimotion" for an example of the performance levels achieved by this technology.) Given such varying video-display performance, the synchronization system must be built to compensate for the variable video-stream performance and still deliver synchronized video and audio data.

Synchronizing Stream Handlers

One of a stream handler's responsibilities is to report its progress to the SSM. In turn, the SSM monitors each stream and provides tolerance checks of a given "slave" stream against a "master" stream. In OS/2, the video stream is a slave stream and the audio stream is the master stream.

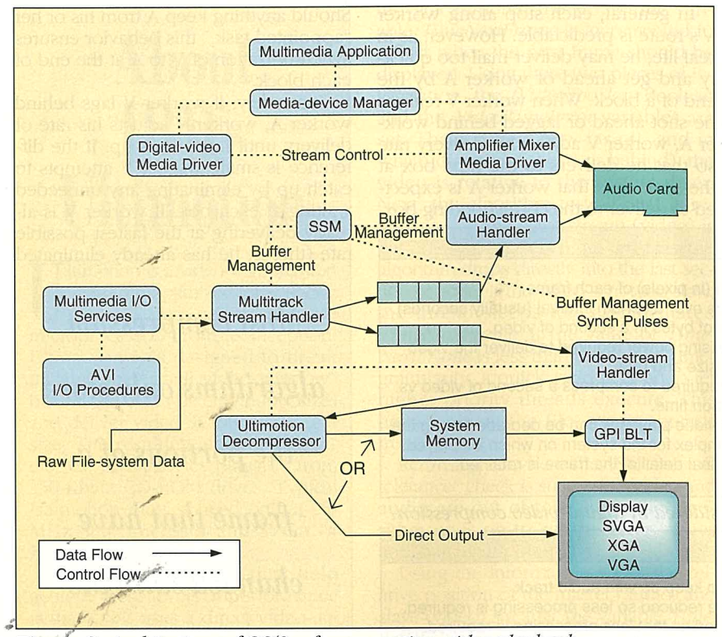


Figure 2: *Architecture of OS/2 software motion-video playback.*

When there's sufficient processing power to handle a movie's frame rate, the video stream handler displays a frame, calculates the time to the next frame, and sets a timer for that duration. As long as the system and audio times remain in tolerance, the system behaves like a free-running system. However, this rarely happens for very long and before you know it, you need adjustments in the video output timing. To acheive synchronization between the two streams, the slave stream adjusts itself to the master stream. For video, this adjustment is made in the calculation of when to display the next frame.

Figure 3 shows a flowchart of the algorithm. However, the details of the algorithm may be better illustrated with an analogy. Consider two postal workers with delivery routes containing the same number of mailboxes. Each postal worker tries to deliver the mail to each box at the same time the other worker delivers the mail to the corresponding box. Postal-worker A (audio) delivers mail at a large apartment complex. The mailboxes have a central location and an efficient and pre-

Digital Video Compression

Digital-video compression is the act of taking a raw digitized image and reducing the amount of storage required to represent the image. There are two compression domains: spatial (or intraframe) compression, which tries to eliminate redundancies within a given frame; and temporal compression, which tries to eliminate redundancies over intervals of time (frames). An algorithm's resulting compression ratio is determined by the degree to which it is able to exploit redundancies and irrelevancies in each of these domains.

There are also two types of compression: lossy and lossless. As its name implies, lossless video compression compacts the source data without losing any of the information it contains. For video data, this means that the compression retains all the image detail. When the image is decompressed, the result is identical to the original. Lossless algorithms are well suited for compressing computer-generated images and are commonly used for storing video animations. To date, however, lossless video algorithms are still too computationally complex for software playback and have limited applicability when the objective is to achieve high compression and frame rates.

Lossy algorithms compact the source data by throwing away data that is redundant from one frame to the next. Examination of a reconstructed frame from any of today's software algorithms reveals the use of this technique. The advantage of this technique is that a respectable representation of the original is retained and it satisfies the requirement for high compression ratios and low computational complexity.

When examining software-compression algorithms, one must consider several characteristics; see Table 1. Because these characteristics are interrelated, you should avoid stressing one at the expense of another. Consider the frame size, which is determined at movie-creation time. Frame size is the width and height (in pixels) that the compression algorithm stores in the movie. In a video digitizer that uses 16-bit color (65,535 colors), the raw data for each frame of a 320×240 movie is 153,600 bytes. At 15 frames per second, the raw data is 15 times that, or 2,304,000 bytes. Assuming 22 Kbytes per second for an audio track, a video-compression algorithm must achieve an average compression ratio of 18:1 to play back at single-speed CD-ROM rates.

The frame rate, determined at movie-creation time, is usually expressed in frames per second. The frame rate determines how smoothly the motion is perceived by the viewer. Studies show that most people dislike frame rates below 12 frames per second. Most movies made for software motion video are 15 frames per second and higher.

Data rate expresses how much bandwidth is required for a movie. A given movie's frame size, frame rate, and compression ratio determine how much space is required for a given interval of time. Usually expressed in bytes per second, the data rate simply expresses the average number of bytes used for one second of movie. A movie's data rate must not exceed the storage device's data rate. If it does, the device won't deliver the data fast enough, and the movie will break up.

A 16-bit color, 320×240 movie running at 15 frames per second takes 2,304,000 bytes (320×240×2×15) for one second of uncompressed (raw) data. Using a compressor that averages 18:1 compression, one second of video data can be reduced to 128,000 bytes. Adding a 22,000-byte audio track, the resulting movie has a data rate of 150,000 bytes per second.

The compression algorithm determines a movie's computational complexity, which is a measure of how much processing power is required to decompress and display a movie's frames. The density at which an algorithm encrypts the data, the technique required to reconstruct each image, and the volume of data per frame determine an algorithm's computational complexity. As an algorithm's playback computational complexity increases, a given processor's ability to decompress it is diminished. To bring it back in balance, the volume of data must be reduced using a smaller frame size, lower frame rate, or both. In this way, a given algorithm can be evaluated on the basis of how large a frame rate and size can be achieved without compromising the appearance of the image.

Using the previous example, we can calculate that the average number of bytes for each compressed frame is 10,000 (150,000 bytes/15 frames per second). Therefore, the decompression algorithm has 1/15 of a second to read the 10,000 bytes, reconstruct the frame, and display it. The higher a movie's frame rate, the less time there is to do this. The larger the frame size, the more pixels have to be displayed. The more dense or complex the compressed data, the more time is required to reconstruct the frame.

Compression complexity determines how an algorithm is used to make a movie. An algorithm that takes longer to compress a frame than it does to decompress a frame is called an "asymmetric," or "off-line" algorithm. Conversely, one that takes equally long to compress as decompress is called a "symmetric," or "real-time" algorithm.

Off-line algorithms first get raw data from a file or frame-stepped device and then compress it. By their nature, these algorithms usually have the best image quality and compression ratios. Real-time algorithms are useful on live video sources that are not controlled one frame at a time. These algorithms compress the data on the fly as it is digitized. The advantage of these algorithms is that they eliminate the need for vast disk storage of raw data. Real-time algorithms usually trade off frame size and compression to achieve reasonable frame rates.

A movie's author determines frame rate, frame size, and data rate at movie-creation time. If played on a system that lacks sufficient power to handle the movie (for example, if the frame rate is too high), the scalability of an algorithm determines what the playback system can do to compensate. The degree to which a movie's characteristics (frame rate, size, resolution, or color depth) can be scaled down at playback time is called its "playback scalability." Table 2 shows how each characteristic of a movie can scale.

Image quality is largely a subjective measure of how well the algorithm retains the details of a movie. As lossy algorithms, today's software motion-video algorithms are perceived differently by different people. However, in general, as image detail increases, one or more of the other characteristics (such as computational complexity) are affected. Again, the overall quality of an algorithm is a function of how well it balances all of these characteristics.

Each video-compression algorithm tries to balance these characteristics to deliver the best video possible in its environment. Which algorithm is best for you depends on which characteristics and environments are most important to you. To some degree, most algorithms let you trade off one characteristic to improve another (for example, reduce the frame size so that you can increase the frame or data rate). Ultimately, before drawing any conclusions, make sure you see a good representation of movies for yourself.

—L.W.

dictable means for mail delivery. Worker A calls in his progress to the main post office (SSM) on a regular basis. Postal-worker V (video) delivers mail in a nearby suburb. This route has rural mailboxes, and the worker must drive from mailbox to mailbox. At the end of each block, postal-worker V calls in (to SSM) and reports his current box number.

In general, each stop along worker V's route is predictable. However, as in real life, he may deliver mail too quickly and get ahead of worker A by the end of a block. When worker V realizes he shot ahead or lagged behind worker A, worker V adjusts his delivery rate so that he delivers to the next box at the same time that worker A is expected to deliver to the corresponding box.

Characteristic	Description
Frame size	Width and height (in pixels) of each frame.
Frame rate	Number of frames over a certain interval (usually seconds).
Data rate	Average number of bytes in a second of video.
Computational complexity	Amount of processing power required to deliver the video at its authored size and rate.
Compression complexity	Amount of time required to compress a second of video vs. its decompression time.
Playback scalability	Degree to which video playback can be degraded when the video is too complex for the system on which it's played.
Image quality	How well the original detail of the frame is retained.

Table 1: *Characteristics that must be considered in digital-video compression.*

Characteristic	How it Scales
Frame Rate	Frames can be dropped to keep up with audio track.
Frame Size	The output window can be reduced so less processing is required.
Resolution	Image detail can be skipped so that less processing is required.
Color Depth	Movies with more colors than are available can be mapped to displays with fewer colors.

Table 2: *Characteristics of a movie and how it can scale during playback.*

Characteristic	Description
(a)	
Frame Size	320×240
Frame Rate	5 frames per second
Data Rate	150 Kbytes per second
Computational Complexity	25-MHz 386
Compression Complexity	Both off-line (8 seconds per frame) and real-time
Scalability	Scales from 65,535 to 16 colors; frame rate: up to the authored rate; frame size: half, normal, and double size
(b)	
Frame Size	640×480
Frame Rate	5 frames per second
Data Rate	150 Kbytes per second
Computational Complexity	25-MHz 486
Compression Complexity	Off-line (5 seconds per frame)
Scalability	Scales from 65,535 to 16 colors; frame rate: up to the authored rate; frame size: half, normal, and double size
(c)	
Frame Size	320×240
Frame Rate	30 frames per second
Data Rate	300 Kbytes per second
Computational Complexity	50-MHz 486DX
Compression Complexity	Off-line (approximately 5 seconds per frame) and real-time
Scalability	Scales from 65,535 to 16 colors; frame rate: up to the authored rate; frame size: half, normal, and double size

Table 3: *(a) Ultimotion's characteristics on a 150-Kbyte-per-second CD-ROM drive for 320×240 frame size; (b) Ultimotion's characteristics on a 150-Kbyte-per-second CD-ROM drive for 640×480 frame size; (c) Ultimotion's characteristics on a 300-Kbyte-per-second CD-ROM drive for 320×240 frame size.*

Should anything keep A from his or her "appointed task," this behavior ensures synchronization of V to A at the end of each block.

Conversely, if worker V lags behind worker A, worker V adjusts his rate of delivery until he catches up. If the difference is small, worker V attempts to catch up by eliminating any unneeded waiting at each box. If worker V is already delivering at the fastest possible rate (that is, he has already eliminated waiting between boxes) and chronically lags behind, a more drastic change in delivery is required. In this analogy, we let postal worker V race down the block and simply toss the mail out the window at each mailbox. Worse yet, we may forget delivery altogether and just drive to the end of the block. (For video, the actual technique used depends on the capabilities of the video decompressor. In either case, the basic idea is to try and drop frames so that the video correlates with the audio.)

Most compression algorithms only store the portions of a frame that have changed since the last frame

Listing One, page 41, is a C implementation of this algorithm using the OS/2 system and synchronization stream manager (SSM) APIs. The routine that calculates the next video-frame decompression time (that is, the next time for worker V to deliver at a mailbox) is called *CalcNextFrameIval*. The input parameters to this routine are pointers to instance structures. The pointer *psib* points to the SSM timing information for this thread. The pointer *pMovie* points to the video stream handler's instance data for the movie being played. Local variables exist for calculating the various error values used by the algorithm as well as flags used for controlling the path through the algorithm.

First, the code gets the current time using the system timer. Based on the movie's authored frame rate, the variable *TimeNextFrame* is incremented to reflect when the next frame should be displayed. Using these two pieces of information, the *fVideoTooSlow* Boolean and the *VideoTimeError* variables are set. These reflect how far off, and in which direction (ahead or behind), the video is relative to the system timer. Next, the algorithm tests if SSM is reporting that the slave stream (video) is out of tolerance. When this test fails, the algorithm drops directly into the last section of code. If the video is not too slow, the thread sleeps for the calculated *TimeNextFrame*. If the value of *TimeNextFrame* is less than the system-timer granularity, a quick yield is done to let higher priority threads execute. This helps prevent the file-system threads from starving.

Returning to where the SSM "out-of-tolerance" check is successful (video and audio are out of tolerance), the algorithm goes on to set the error values and synchronization flags.

Using the information about the relative position of the master (audio) and slave (video) streams, the variable *TimeNextFrame* is calculated. It's important to note that at this point, the algorithm works to force the video in sync with the audio and not in sync with where it computes it "should" be.

The comments in the code detail the precise conditions tested and the way *TimeNextFrame* is calculated. However, regardless of which stream lags behind the other, the algorithm uses the same scheme. When video lags behind audio, *TimeNextFrame* is set behind the current time. This has the effect of making the video play faster. Conversely, when the audio lags behind video, *TimeNextFrame* is set ahead of the current time, which has the effect of slowing down the video. The amount by which *TimeNextFrame* is set ahead or behind the current time is always calculated to be the difference between the audio time and the system timer. In addition to forcing the video to synchronize to the audio, this cross check also adjusts for possible differences in the timing sources. (For example, SSM and the system timer use different physical timers, and the two clocks drift.)

Once *TimeNextFrame* is recalculated, the algorithm drops into the same code discussed earlier that takes action based on the local flags and error values. If it isn't time to display the next frame, the

About IBM's Ultimotion

Ultimotion is a video-compression algorithm optimized for software playback on a general-purpose microprocessor. From its inception, Ultimotion was designed to break through the "small-video-window" barrier (160×120 pixel window size) and deliver video at four times that size. The resulting algorithm delivers 320×240 movies playable from 150-Kbyte CD-ROM drives. Typical Ultimotion movies need only a 25-MHz 386 processor and SVGA or XGA display adapter.

One of the factors that help achieve these levels of performance is that OS/2 uses a direct video-access technique now beginning to emerge in other systems. When supported by a display adapter's device driver, the video decompressor is given direct access to the display adapter. The system automatically detects the presence of the support and uses it without the knowledge of the decompressor. Using this high-speed access, 486-based machines are able to play Ultimotion's larger 320×240 movies at 30 frames per second.

Ultimotion is a software-only, video-compression algorithm that averages 18:1 compression. It uses both spatial and temporal compression and was designed to run on processors as low as a 25-MHz 386. On a single-spin CD-ROM drive (150 Kbytes per second), it exhibits the characteristics summarized in Table 3(a). Table 3(c) shows the characteristics on a double-spin CD-ROM drive (300 Kbytes per second).

Ultimotion delivers a respectable frame size and frame rate at relatively low data rates and processing power. Its organization of the compressed data enables efficient output display, clipping, doubling, and halving of the movie's frame size. Ultimotion movies can be created using either off-line or real-time algorithms across a wide range of frame sizes and frame rates.

—L.W.

See you at COMDEX, Booth #A-1958
Video Compression is not simple.
But with our libraries, you'll think it is.
MPEG
JPEG
Px64
VDS™ Kit
Video Decompression Source
The VDS Kits are software libraries which fully implement the complete video compression standards. That leaves you free to do what only you can do: create the best mulitmedia application possible, using photos and video. The VDS Kits achieve up to 200:1 compression on all major systems including DOS, Windows, NT, UNIX Macintosh and many popular DSP chips. And at our special introductory price of $599, it's just a drop in the bucket. With OEM, distributors, and source code licenses available, the VDS Kits are just about the easiest way to add video to your applications.
Performance Computing INCORPORATED
(503) 297-2292
or fax: (503) 297-0878
VDS Kit Sales
1815 S.W. Marlow Ave, Suite 206
Portland, Oregon 97225
CIRCLE NO. 874 ON READER SERVICE CARD

thread sleeps for the calculated interval. If the time is past, the algorithm stores this information in the movie-instance structure for use by the frame-dropping algorithm.

Dropping Frames to Catch Up

At first glance, dropping frames to make video "catch up" seems simple and straightforward. However, most software algorithms use temporal compression and only store "delta frames"—those portions of a frame that have changed since the last frame. If these frames are dropped, portions of the movie that should have changed aren't changed from frame to frame and unpleasant visual artifacts are displayed. To counter this, the compression algorithm frequently compresses the entire frame and inserts it into the video stream. This intraframe compressed frame is called an "I-frame." When displayed, I-frames repaint the entire video frame and repair any artifacts.

Since frame dropping depends on the compression algorithm, the frame-dropping logic in the stream handler defers the actual drop processing to the decompressor. If the compressed data stream cannot tolerate frame dropping, the decompressor simply ignores the information and the system continues as best it can.

Summary

A fully threaded system provides efficient tools to process multimedia data. These tools effectively help address multimedia-related problems in areas such as buffer management, data movement, data filtering, and synchronization. As a platform for multimedia applications, OS/2 provides a rich set of synchronization and video-output services. Through the use of threads, system-timing services and OS/2's multimedia synchronization services, the synchronization algorithm presented here can provide an independent video-timing mechanism for when an audio track is not present. It can also correct for the following: clock drifts in system and audio timers, dynamically and statically varying display subsystem efficiency, chronic video-synchronization loss, and catastrophic audio failure. It is an example of how multiple timing services can be combined to provide a solution to an age-old problem.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

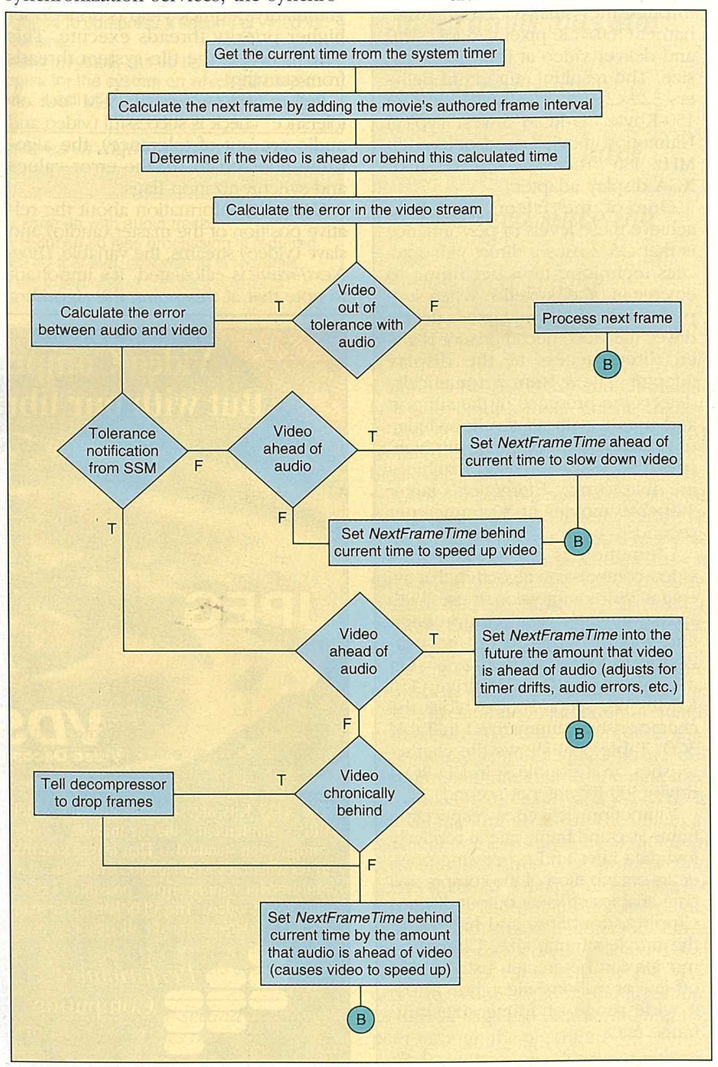

Get the current time from the system timer
Calculate the next frame by adding the movie's authored frame interval
Determine if the video is ahead or behind this calculated time
Calculate the error in the video stream
Video out of tolerance with audio
T
F
Calculate the error between audio and video
Process next frame
B
Tolerance notification from SSM
F
Video ahead of audio
T
Set NextFrameTime ahead of current time to slow down video
F
Set NextFrameTime behind current time to speed up video
B
T
Video ahead of audio
T
Set NextFrameTime into the future the amount that video is ahead of audio (adjusts for timer drifts, audio errors, etc.)
B
F
Tell decompressor to drop frames
T
Video chronically behind
F
Set NextFrameTime behind current time by the amount that audio is ahead of video (causes video to speed up)
B

Figure 3: *Logic flow of audio/video synchronization algorithm.*

"Plant a tree.... Trees for America"

Join me and plant a tree. Trees for America...and for the earth. For your free brochure, write:

Trees for America, The National Arbor Day Foundation, Nebraska City, NE 68410.

John Denver for
The National Arbor Day Foundation

Listing One *(Text begins on page 34.)*

```
RC CalcNewFrameIval  ( PSIB         psib,
                       PMOVIE_STR   pMovie )
{
   LONG     AudioSynchError;
   LONG     VideoTimeError;
   LONG     lmsTimeError;
   BOOL     fVideoTooSlow = FALSE;
   BOOL     fVideoBehindAudio = FALSE;
   BOOL     fSynchPulse = psib->syncEvcb.ulSyncFlags&(SYNCPOLLING+SYNCOVERRUN);
   ULONG    CurrentTime;
   MMTIME   mmtimeMaster = psib->syncEvcb.mmtimeMaster;

   // get current time
   DosQuerySysInfo (QSV_MS_COUNT,QSV_MS_COUNT,&CurrentTime,sizeof(ULONG));

   // Update frame time
   pMovie->TimeNextFrame += pMovie->FrameInterval;


      //************************************************
      // Determine if the video is ahead or behind
      // the frame rate specified for this stream.
      //************************************************
      if (CurrentTime <= pMovie->TimeNextFrame) {

         //*******************************************************
         // Video is ahead according to system clock
         // Compute Video Error
         //*******************************************************
         VideoTimeError = (pMovie->TimeNextFrame - CurrentTime);
         fVideoTooSlow = FALSE;

      } else {
         //*******************************************************
         // Video is behind according to system clock
         // Compute Video Error
         //*******************************************************
         VideoTimeError = (CurrentTime - pMovie->TimeNextFrame);
         fVideoTooSlow = TRUE;

      } /* endif - who is ahead? */


      //**************************************
      // Is SSM reporting "Out of Tolerance"
      // Check for a Synch Pulse
      //**************************************
      if (psib->fSyncFlag == SYNC_ENABLED) {

         if (fSynchPulse && mmtimeMaster) {
               pMovie->ulSynchPulseCount++;   /* Accumulate count of sync pulses


               /*************************************************************/
               /* Is SSM reporting Video Behind Audio?
               /*************************************************************/
               if (mmtimeMaster >
              (psib->syncEvcb.mmtimeStart + psib->syncEvcb.mmtimeSlave))
               {  // Video is behind audio


                  fVideoBehindAudio = TRUE;

                  // Compute Audio error
                  AudioSynchError = mmtimeMaster -
                        (psib->syncEvcb.mmtimeStart + psib->syncEvcb.mmtimeSlave);

                  // Is Video behind according to system clock?
                  //
                  if (fVideoTooSlow) {
                     if ( AudioSynchError > VideoTimeError ) {
                        // Set the next frame time behind the current time
                        // so that the delta to the cur time = SSM Audio Error
                        // This will cause the video to speed up

                        pMovie->TimeNextFrame -=
                        ( AudioSynchError - VideoTimeError );


                     } else {
                        // Set the next frame time ahead of the current time
                        // so that the delta to the cur time = SSM Audio Error
                        // This will cause the video to slow down
                         pMovie->TimeNextFrame +=
                                ( VideoTimeError - AudioSynchError );

                     }

                  } else { // Video OK by System Clock but SSM reports otherwise
                           // Set the next frame time behind the current time so
                           // that the delta to the current time = SSM Audio Error
                           // This will cause the video to speed up
                     pMovie->TimeNextFrame -= (AudioSynchError + VideoTimeError);

                  } /* endif */

               } else {  // SSM reports Video ahead of Audio

                  fVideoBehindAudio = FALSE;
                  AudioSynchError =  (psib->syncEvcb.mmtimeStart +
                  psib->syncEvcb.mmtimeSlave) -
                                       mmtimeMaster;

                  // Is video behind according to system clock?
                  if (fVideoTooSlow) {
                     //*************************************************************
                     // Video is behind according to system time, but video is
                     // running ahead of the audio (the audio must have started
                     // late or somehow broken up, fallen behind and can't get up
                     //*************************************************************
                     // Set the next frame time ahead of the current time
                     // so that the delta to the cur time = SSM Audio Error
                     // This will cause the video to slow down
                     pMovie->TimeNextFrame += AudioSynchError + VideoTimeError;

                  } else {  // Video ahead according to system clock and SSM
                     //*************************************************************
                     // Video is keeping up or is ahead according to system time
                     // AND video is running ahead of audio.
                     //*************************************************************

                     if ( AudioSynchError > VideoTimeError ) {
                        // Video is further ahead than system clock indicated
                        // Set the next frame time ahead of the current time
                        // so that the delta to the cur time = SSM Audio Error
                        // This will cause the video to slow down
                        pMovie->TimeNextFrame += AudioSynchError-
                                                 VideoTimeError;
                     } else {
                        // Video not as far ahead as system clock indicated
                        // Set the next frame time behind the current time
                        // so that the delta to the cur time = SSM Audio Error
                        // This will cause the video to speed up
                        pMovie->TimeNextFrame -= VideoTimeError-
                                                 AudioSynchError;
                     }


                  } /* endif */

               } /* endif */

               psib->syncEvcb.ulSynchFlags = 0;

               //*************************************************************
               // Recompute video time error based on updated TimeNextFrame
               //*************************************************************
               if (CurrentTime <= pMovie->TimeNextFrame) {
                  VideoTimeError = (pMovie->TimeNextFrame -
                  CurrentTime);
                  fVideoTooSlow = FALSE;

               } else {
                  VideoTimeError = (CurrentTime -
                  pMovie->TimeNextFrame);
                  fVideoTooSlow = TRUE;

               } /* endif - time exceeded. */

            } /* endif - SSM reporting "Out of Tolerance" */
         } /* endif - Listening to SSM */

         //************************************
         /* Take action based on whether the
         /* video is running ahead or behind
         //************************************
         if (!fVideoTooSlow) {

            lmsTimeError = VideoTimeError / 3;  //Convert error to milliseconds
            if (lmsTimeError > 32L) {
               // Block till next time to display a frame
               DosSleep(lmsTimeError);
               pMovie->ulLastBlockTime = CurrentTime;
            } else {
               // Too close to next frame time for system clock to be used
               // Be good and yield to higher priority thread if one around
               DosSleep(0);
               pMovie->ulLastBlockTime = CurrentTime;
            }

         } else {

            /*******************************/
            /* Drop some frames if behind!  */
            /*******************************/
                     :
                     :
         } /* endif - video too slow */

      // Update Frame Count
      pMovie->ulFrameNumber++;


      return (NO_ERROR);
}
```

End Listing

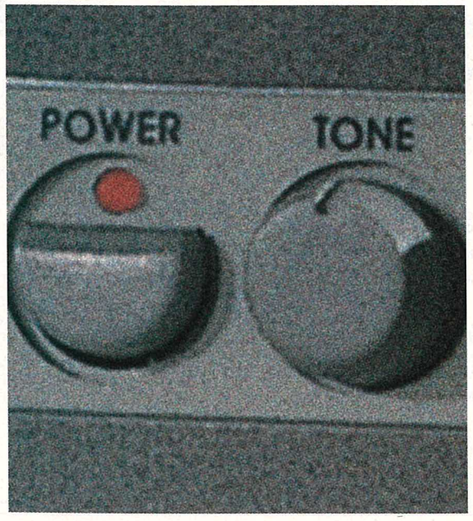
Programming the QUANTUMdsp

Downloadable microcode makes softcoding a reality

Charles A. Mirho

Many conventional multimedia boards are *hard function* in that their capabilities are defined by ROM chips (or ROM embedded in microcontrollers) planted on the board at the factory. The problem with the hard-function approach is that technology, particularly in the areas of audio compression and telecommunications, marches ruthlessly on. How state-of-the-art is that 8-bit SoundBlaster board you purchased two years ago, or that three-year-old modem? New compressions offering improved quality and higher ratios are constantly being invented. Standards evolve. Modem and fax technology advances toward higher bit rates. As a result, hard-function hardware becomes less than state-of-the-art within months of its manufacture. The QUANTUMdsp board from Communication Automation & Control offers a solution to this dilemma. The microcode that other multimedia and telephone boards hardcode into ROM is encapsulated into disk files. These disk files are downloaded to the board's local RAM as needed. The QUANTUMdsp's capability at any given time is defined by the contents of this RAM. Therefore, the board can be decoding a JPEG image while simultaneously playing MPEG-coded audio and rotating multiple 3-D wireframes. Or it can be synthesizing MIDI while answering the telephone, simply by replacing the contents of RAM with a new set of functions. The great advantage of this approach is that pieces of the microcode can be combined and configured on the fly. What's more, updating the microcode disk files updates the board's capabilities. For instance, the initial beta version I had contained a disk file with microcode to emulate a 9600-baud modem. Within weeks I received a floppy disk with new microcode to emulate a 14.4-baud modem. I copied the new file over the old one, and I had a working 14.4-baud modem running from the Windows Terminal program.

Charles is a consultant specializing in multimedia and telephony. He can be reached on CompuServe at 70563,2671.



The QUANTUMdsp board supports advanced audio-compression formats such as MPEG, G.722, G.728, and subband coding, as well as the more well-known µLaw, aLaw, and ADPCM compressions. The board includes a general-MIDI synthesizer, baseline 24-bit JPEG decoder, 14.4-Kbps analog modem, Class I fax, speaker-independent voice recognition, and a fast, graceful means of rotating 3-D wireframes. And, as just mentioned, many of these are available simultaneously.

Sound Familiar?

Historically, this isn't a new approach. Desktop printers followed a similar evolution during their early stages. Over the last 15 years, printers evolved away from hard function. Early printers had only a built-in set of fonts (still true of low-end printers). When an application used the printer, it was stuck with these internal fonts. If, later on, the application required better fonts, the user had to purchase a new printer, or possibly upgrade the printer's ROM chip to include the new fonts. Then laser printers came along, and everything changed. If an application required a font not included in the printer's internal set, it simply downloaded a suitable "soft font" to the printer. This flexibility was a strong influence in the rise of desktop publishing.


```
FifoSize:   40000               /* size of flow-control buffer */
Local: FlowBuffer FifoSize      /* declare flow control buffer */
Extern: AudOut_8_1              /* output channel */
fta( ILevel)
{
        fin     FlowBuffer
        aout    AudOut_8_1
}
```

Example 1: *Script to play linear, 16-bit audio at 8000 samples per second.*

How it Works

A library of microcode files (the manual calls them "modules") are stored in a separate directory. These modules can be combined sequentially using a simple scripting language. Example 1, for instance, shows a script file for playing linear, 16-bit audio at 8000 samples per second. Linear data exists in the raw, 16-bit native format of the A/D converter that sampled it. No special processing is required to play linear data; it's moved directly from the source to the speaker. This is rarely the case — audio data is usually coded, which means each sample has been compressed. (Without compression, audio files can consume 2 to 50 times more space on a hard drive.) The script in Example 1 offers some insight into the board's inner workings. The first two lines declare a *flow-control buffer.* A Windows application can get a pointer to this buffer and fill it with audio data from a disk file. (We will see how this is done in a moment.) The flow-control buffer is needed for two reasons. First, the Windows application runs on the PC (Intel) processor under cooperative multitasking, which means its behavior is nondeterministic. That is, it is subject to unpredictable delays. The module (*fta* in the script) is responsible for feeding data to the speaker at exactly 8000 samples per second. That means having a steady supply of audio data on hand at all times; delays are unacceptable, unless you enjoy gaps and hiccups in your audio. The module cannot be left to the mercy of the Windows program, since the Windows program is at the mercy of other Windows programs, ISRs, the Windows memory manager, and so forth.

The second reason for the flow-control buffer is a bit more complex and involves signal theory. Essentially, the problem is that it is most efficient for programs to read large chunks of data at a time from disk files. The Windows program, for the sake of efficiency, would like to read between 4 Kbytes and 32 Kbytes at a time from the audio file. The DSP module, on the other hand, must execute in real time. (The playing of audio is inherently real-time in nature, since we hear sound in real time.) For this reason the DSP module can only deal with tiny chunks of data at any one time (typically between 50 and 3000 bytes). The flow-control buffer lets both the Windows application and the DSP module have it their way — the app feeds large chunks of data into one end of the buffer at a rate it is comfortable with, while the DSP module takes data in small blocks from the other end at its own rate. For this to work, the buffer needs two ends, a head, and a tail. Thus the flow-control buffer is implemented as a FIFO data queue.

The FIFO is a circular buffer with associated read and write pointers. The read pointer marks the spot in the FIFO where the next read operation will find data. The write pointer marks the spot where the next write operation will put data. In Figure 1, the area indicated in light green contains data. Reading this data increments the read pointer. Writing more data increments the write pointer. The FIFO is full when the write pointer is one position to the left of the read pointer. The FIFO is empty when the read and write pointers are equal.

The third line in the script declares a *channel* — a connection to an external device (such as a speaker, microphone, or telephone line). Channels have two other characteristics besides the device type: the *sampling rate* and the *device number.* The sampling rate is simply the number of samples per second that will pass through the channel. The device number specifies which device of the specified type you wish to connect to. If two speakers are connected to the board, you could direct the audio to the first speaker by setting the device number to 1. A declaration of the channel in the form *device.type_sample.rate_device.number* defines all of these characteristics: 8000 samples per second flow over the channel to the first speaker connected to the board. Therefore, *AudOut_8_1* in Example 1 specifies an audio output device (a speaker) with 8000 samples per second, which connects to the first speaker (there may be more than one).

Finally, the module itself is declared. The module *fta* is nothing more than a piece of DSP microcode. The definition of modules in the script is object oriented; each module is treated as a "black box" with inputs, outputs, and possibly, controls. The *fta* module is simply designed to take "bites" of its input and move them to its output. The input of *fta* is connected to the flow control buffer, and the output is connected to the channel. *fta* bites off 80 samples at a time from the flow-control buffer and moves them out over the channel where they reach the speaker to be perceived as sound. The DSP always reads 100 blocks of data per second for a total throughput of 8000 samples per second.

Taken in its entirety, the script defines the flow of audio data from the Windows application to the first speaker connected to the board. Graphically, the data flow looks like Figure 2.

The module library is full of simple, useful modules like *fta.* The modules can be combined sequentially in a script to form more-complex multimedia functions. However, such examples are beyond the scope of this article.

Change is Easy

You may be wondering if all of this scripting is worth the trouble. After all, playing 16-bit uncompressed audio at 8000 samples per second is hardly a

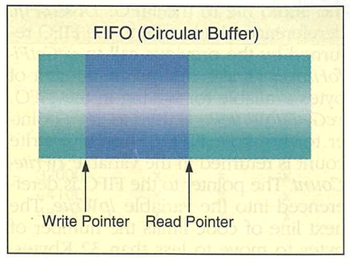


Figure 1: *The flow-control buffer.*

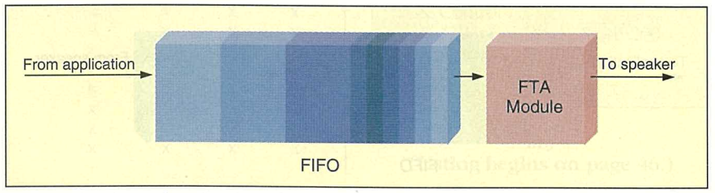


Figure 2: *Flow of audio data from a Windows application to the first speaker.*

monumental feat. Example 2, however, shows how simple it is to modify the script for playing audio at 44,100 samples per second. The only difference is in the channel declaration, *AudOut_44_1*. The number 44 is shorthand for 44,100, just as 8 is shorthand for 8000.

Suppose you wanted to record audio at 44,100 samples per second instead of playing it. You simply modify the script to reverse the flow of data; see Example 3. There are two changes to the script in Example 3. The first is the channel declaration, which changes the device type from *AudOut* (specifying a speaker) to *AudIn* (specifying a microphone). The second difference is that the *fta* module has been replaced with the module *atf*, which works like *fta* but in reverse; it accepts blocks of data from the channel and moves them to the flow-control buffer. Graphically, the flow of data looks like Figure 3. To add or remove compression from the audio stream, a coder/decoder module could be added in series with either the *atf* or *fta* module. The module library is full of useful coders and decoders.

Some C Required

Returning to the example of playing 16-bit, linear audio at 8000 samples per second, on the Windows side, a C program is required to read the data file and move the audio data into the flow-control buffer so that it can be sent out over the channel. Listing One, page 46 shows the complete program. The program has the structure of a DOS program and is compiled and linked for QuickWin. This isn't a requirement, but I use it for the benefit of DOS programmers making the transition to Windows. QuickWin programs are much easier to follow than conventional Windows programs, while still illustrating all the important concepts.

Each board contains a single floating-point 32-bit DSP running at 55 MHz

The program begins with a list of necessary header files. In addition to the standard C header files, the file vclib.h is included. This header defines the API to the multimedia board. You can easily spot board-specific functions in the example; they are all prefixed by the letters *vc*.

```
FifoSize:   40000              /* size of flow-control buffer */
Local: FlowBuffer FifoSize     /* declare flow control buffer */
Extern: AudOut_44_1            /* output channel */
fta( ILevel)
{
        fin     FlowBuffer
        aout    AudOut_44_1
}
```

Example 2: *Modifying the script in Example 1 to play linear, 16-bit audio at 44,100 samples per second.*

```
FifoSize:   40000              /* size of flow-control buffer */
Local: FlowBuffer FifoSize     /* declare flow control buffer */
Extern: AudIn_44_1             /* input channel */
atf( ILevel)
{
        ain     AudIn_44_1
        fout    FlowBuffer
}
```

Example 3: *Script to record linear, 16-bit audio at 44,100 samples per second.*

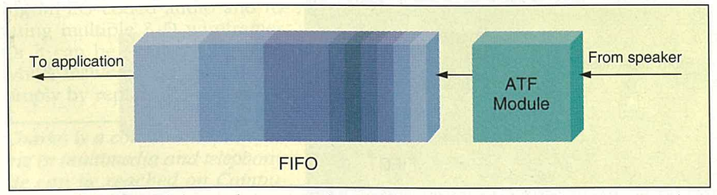


Figure 3: *Flow of data when recording audio at 44,100 samples per second.*

The first call is to function *vcAddTask*, which loads the script file. The microcode from any disk modules in the script are downloaded to the board at this point. The 40,000-byte flow-control buffer defined in the script is also allocated. The first parameter to *vcAddTask* is the DSP number on which the script will execute. Each board contains a single floating-point 32-bit DSP running at 55 MHz. This should be enough for all but the most intensive applications. If you need more power, however, up to four boards can be added in a single machine. The second parameter is the name of the script file to load. The function returns a handle which will be used to identify this script in future function calls. (A single program can load many scripts simultaneously.)

At this point the script is loaded and idle. The example calls the functions *vcGetFifoHandle* and *vcInitFifo*. The function *vcGetFifoHandle* returns a handle to the FIFO buffer, just as the Windows memory-allocation functions return handles to memory blocks. This handle will be dereferenced shortly when it is time to move data from the file into the FIFO. The call to *vcInitFifo* simply sets the FIFO read pointer equal to the value of the FIFO write pointer, indicating that no data is available in the FIFO (the empty state).

Next the data file is opened. This data file contains audio data sampled at 8000 samples per second. After opening the data file, Listing One calls the function *DiskToFifo*. This isn't a board function but rather a local function for moving data from the audio file to the FIFO. We will see how the function does this in a moment; for now, suffice it to say that when the function returns, the FIFO is loaded with data and ready to go. The script is loaded and sitting idle with data in the buffer. All that remains is to call *vcStartTask* to get things started. *vcStartTask* takes a single parameter, the script handle returned by *vcAddTask*.

Inside *DiskToFifo*

Function *DiskToFifo* moves data from the audio file to the FIFO. *DiskToFifo* dereferences the handle to the FIFO returned by the previous call to *vcGetFifoHandle*. It also returns the number of bytes available for writing in the FIFO. *vcGetFifoWritePtr* is used to get a pointer to the write FIFO. The FIFO write count is returned in the variable *lWriteCount*. The pointer to the FIFO is dereferenced into the variable *lpWrite*. The next line of code limits the number of bytes to move to less than 32 Kbytes. This isn't a limitation of the board software but rather a choice in this example to avoid huge data moves that can cause problems in the Intel segmented architecture.

Data is read directly from the data file into the FIFO using the standard C language *read* function. Since data is read directly into the FIFO this way, the write count returned by *vcGetFifoWritePtr* must be the number of *consecutive* bytes available for writing in the FIFO (remember, FIFOs are circular buffers). Obviously, a function like *read* is "unaware" of the circular nature of the FIFO buffer, and so it is up to the program to supply the number of consecutive bytes.

If an end-of-file condition is reached, indicating that all the data in the file has been moved to the FIFO, the read count is checked using the formula *lReadCount &=~0x3L*.

This formula rounds down to the nearest multiple of four bytes. This is necessary because the last read may have reached an end-of-file, so the actual read count is something less than the number of bytes requested. For example, if 0x7fff bytes were requested but only 1001 bytes remained in the file, then the read count will be 1001. This number must be rounded down to the nearest multiple of four bytes: *1001 &=~3L0* becomes *1000*. (An annoying anomaly of the FIFOs — they must be a multiple of four bytes in size, and must be read and written in multiples of four bytes.) Unfortunately, this discards the last byte in the file but it's not usually a problem when playing audio files. If it is essential to preserve the last byte in the file, then pad bytes can be added, making the total read a multiple of four.

Even after the entire file is read into the FIFO, it isn't yet safe to shut down and exit the program. The entire file has not been played until the FIFO is empty. The flag *donewriting* is set to indicate that the file has been read into the FIFO and we are now waiting for the FIFO to play out. An *if* statement at the top of the function checks if the FIFO has played out by calling the function *vcGetFifoReadCount*. The FIFO read count will be 0 when the FIFO is empty, indicating that all data has been played.

The last statement in *DiskToFifo* is a call to the function *vcUpdateFifoWritePtr*. This function updates the read and write pointers for the FIFO. The FIFO pointers are not updated automatically when the FIFO is read or written, because there is no way for the board software to know how many bytes the standard C *read* function moved into the FIFO.

The *main* function calls *DiskToFifo* from a tight loop until the entire file has been played. When *DiskToFifo* returns 0, indicating that all data has been played, *main* calls *vcDeleteTask* with the handle to the script file. This unloads the script and frees the memory allocated for the FIFO.

Support for Standards

Developers who prefer standards will be happy to know that the board supports all MPC functions for the playing and recording of wave audio, as well as for MIDI (synthesis only). Technically, the QUANTUMdsp is not MPC compliant because it does not include a MIDI port. MIDI output messages are directed to the board's general-MIDI synthesizer (also a downloadable microcode file). I tested the board with both Sound Recorder and Media Player and both worked well. Table 1 shows the linear wave-audio formats supported by the board.

Most of the audio formats can mix with one another in any combination. That means two or more audio files of different sampling rates and sample size can play simultaneously. Try that with a Soundblaster! Sample rate-converter modules in the microcode library allow two or more audio streams with different underlying sampling rates to mix on the same output channel. (The sample rate-converter modules are inserted automatically when the board software detects a format conflict.) Supported audio coder/decoders are G.722, G.728, MPEG Layer Two, ADPCM, µLaw, aLaw, and subband.

Analog modem and fax (Class I) capabilities are available by replacing the standard Windows COMM driver with the board's COMM driver. Once the driver is installed, the Windows Terminal program can be used to dial from 14.4 kbps right on down to 300 bps. Any Windows communication program that uses the standard COMM driver will work as well. Fax programs which use the Windows COMM driver (such as BitFax from BIT Software) also work at 9600 baud. Demo programs for 3-D wireframes, 24-bit (16-million-color) JPEG still-image decompression, and an audio jukebox are included.

Conclusion

While the microcode library supports an impressive set of audio functions, there's relatively little in the way of image compressions (only JPEG). The lack of a MIDI or joystick port are certainly drawbacks for the music-composition and game markets. Motion video isn't supported, but a video daughterboard is planned for the future.

Still, the high-powered set of audio compressions are well suited to voice mail and teleconferencing applications. The fast JPEG decoder and MPEG audio compression should be very useful in top-end presentation packages. The telephone and fax features rival similarly priced, dedicated communication boards. But probably the most important feature of the board is the fact that the buyer is not committing to today's multimedia standards. As standards evolve, as compression formats improve, and modem bit rates move up, you can expect updates in the form of disk files (courtesy of AT&T and third parties). This should double or possibly triple the useful life of the board over more-conventional hard-function aproaches.

Sample Rate	Mono PLAY 8 bit	Mono PLAY 16 bit	Stereo PLAY 8 bit	Stereo PLAY 16 bit	Mono RECORD 8 bit	Mono RECORD 16 bit	Stereo RECORD 8 bit	Stereo RECORD 16 bit
8000	x	x	x	x	x	x	x	x
11025	x	x	x	x	x	x	x	x
16000	x	x	x	x	x	x	x	x
22050	x	x	x	x	x	x	x	x
24000	x	x	x	x	x	x	x	x
32000	x	x	x	x	x	x	x	x
44100	x	x	x	x	x	x	x	x
48000	x	x	x	x	x	x	x	x

Table 1: *Linear wave audio formats supported (almost all formats can mix).*

For More Information

Communication Automation & Control
1642 Union Blvd., Suite 200
Allentown, PA 18103
800-367-6735

DDJ

(Listing begins on page 46.)

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**

fractal

image

compression

Fractal image compression. The technology that inspired *PC Magazine* to proclaim "The advantages of fractal compression are unmatched." The technology selected for Microsoft®'s breakthrough CD-ROM Encyclopedia, Microsoft® *Encarta*.

This technology is now available to Windows 3.x developers in **POEM ColorBox**, which provides DLLs for:

- ultra-high compression;
- resolution independent images;
- fast decompression; and
- superior 24-bit color rendering.

For multimedia, archiving or other imaging applications, **ColorBox** is comprehensive, easy-to-use and at $995, suprisingly affordable.

Call today for more information. Learn why **ColorBox** is the solution to your image compression needs.

Iterated Systems
5550-A Peachtree Pkwy., Suite 650
Norcross, GA 30092
Tel: 1-800-4FRACTL
Fax: (404) 840-0806

CIRCLE NO. 767 ON READER SERVICE CARD

TIME ARTS **SIGNATURE**™
GRAPHICS CLASS LIBRARY

Draw on Our Experience!

Add sophisticated 2D graphics to your C++ applications in a fraction of the time

- MS Windows (Borland C++ compiler)
- Sun, SGI (Mac version in development)
- Fully object-oriented architecture
- Fully extensible classes
- Reusable objects
- Comprehensive graphics model
- Platform independence

Signature at a Glance

- Antialiasing
- Opacity patterns
- Matte compositing with filters
- Transformations
- Image processing
- Multiple color systems
- Color mapping
- Scalable fonts
- Vector/Raster integration
- Resolution independence

Time Arts

800 959-0509
707 576-7722
Fax: 576-7731

1425 Corporate Center Parkway
Santa Rosa, CA
95407-5434

Call today to learn how our eleven years of graphics know-how can go to work for you!

CIRCLE NO. 886 ON READER SERVICE CARD

Listing One *(Text begins on page 42.)*

```
/*  QuickWin Audio Player */
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <io.h>
#include <errno.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <fcntl.h>
#include <vclib.h>      /* VCAS function prototypes */

int DiskToFifo(long hf, int fd);
static int fd, tidPLAY, tidADA;

main()
{
static long hf;
long hparam;

        if( vcAddTask( 1, "PLAY",&tidPLAY) < 0)  /* load script */
                return -1;
        if( vcGetFifoHandle( tidPLAY, "FlowBuffer", &hf) < 0)
                                        /* get handle to flow control buffer */
                return -1;
        if( vcInitFifo( hf) < 0) /* initialize (zero out) flow control buffer*/
                return -1;
        fd=open( "HELLO.L8", O_BINARY|O_RDONLY); /* open the audio data file */
        if(fd == -1) printf("Cannot open HELLO.L8\n");

        printf ("Hit return...\n");
        getchar();

        DiskToFifo( hf, fd);        /* put some data in the FIFO */
        if( vcStartTask( tidPLAY) < 0)  /* start playing the audio file */
                return -1;
        while(DiskToFifo(hf,fd)==1); /* play until done */
        vcDeleteTask (tidPLAY);
        close(fd);                  /* close input file */
}

int DiskToFifo(long hf, int fd)
{
long    lReadCount, lWriteCount, *lpWrite, li;
static  donewriting = 0;

        if (donewriting)
        {
            vcGetFifoReadCount( hf, &li);  /* if DSP has emptied the FIFO... */
            if(li==0L) return(0);          /* then quit */
            return 1;
        }
        printf(".");            /* print something to indicate activity */
        vcGetFifoWritePtr( hf, &lWriteCount, &lpWrite);
                                        /* get FIFO write pointer */
        if(lWriteCount > 0x7fff) lWriteCount=0x7fff;
                                /* limit data moves to 32K */
                                /* read the disk directly into the FIFO */
        lReadCount= read( fd, (void*)lpWrite, (unsigned int)lWriteCount);
        if(lReadCount < lWriteCount)  /* if disk is getting empty... */
        {
                lReadCount &= ~0x3L;            /* ensure 32-bit transfer */
                donewriting = 1;
        }
        vcUpdateFifoWritePtr( hf, lReadCount); /* update FIFO indices */
        return(1);
}
```

End Listing

Our DSP Development Tool Kits outperform the competition in every way.

Fixed. $499

Floating. $995

Take your pick. Because whether you're looking for fixed or floating-point DSP, when it comes to price and performance our EZ-KITs are clear winners. They're complete packages, giving you everything you need to evaluate and develop your DSP application. In fact, they're so complete and easy to use they'll actually help you get your designs to market faster. Especially with features like an Applications Textbook with a full library of source code, a simulator for debugging and an EZ-LAB™ Evaluation Board.

The ADSP-2100 family handles application diversity while preserving code compatibility. At 0.58 ms 1024-pt complex FFT, the ADSP-21000 family offers the highest floating-point performance available.

We also outperform the competition on compatibility. The ADSP-2100 family of code-compatible fixed-point DSP processors lets you easily port code from the low-cost ADSP-2105 to the fastest ADSP-2115, or the 3.3 V ADSP-2103. And the ADSP-21020 EZ-KIT adds 32- and 40-bit floating point to the line. So when it comes to the DSP tool kit with the most going for it, any way you look at it the answer is Analog Devices. For more information or to order a kit by phone, call us at (617) 461-3771.

ANALOG DEVICES

Analog Devices, Inc., One Technology Way, P.O. Box 9106, Norwood, MA 02062-9106. Distribution, offices and application support available worldwide.
EZ-LAB is a trademark of Analog Devices, Inc.

CIRCLE NO. 884 ON READER SERVICE CARD

Animation with the Windows GDI

Saving and restoring bitmapped images within a window

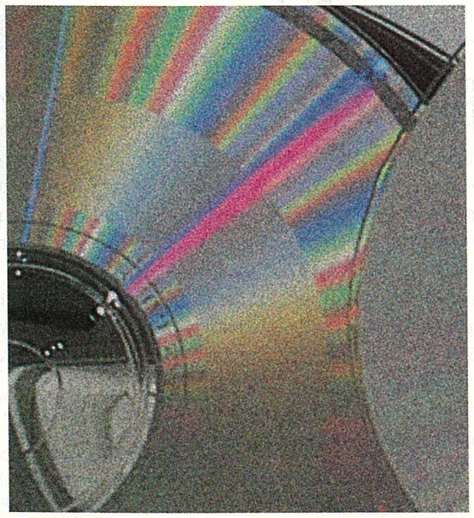
Joe Sam Shirah

Animation with Windows presents some special challenges, as I discovered in a recent project that involved an animation sequence in a multiprogram-database application. The animation was similar to the classic bouncing ball, moving a nonrectangular bitmap over a previously painted background. A requirement of the project was that the animation should work even if the background changed, and it had be independent and reusable enough to easily integrate into other programs. I soon found that my standard library let me down, and I was off on one of those deductive journeys for which Windows so freely offers a ticket to ride.

This article describes some of the quirks…er, features in Windows that, if ignored, can cause graphical graffiti to erupt all over your beautiful displays. Some of the hurdles include performing bit-block transfers (bitblts) in a window and overcoming Windows' optimization procedures when repainting the screen. I'll also address smoothness in larger animations and present a program, MOVBMP, that shows you how to put it all together.

Animating Bitmaps

Most paint programs as well as the *BitBlt* function, which is our primary tool for manipulating images, require rectangular bitmaps. In general, to display nonrectangular images, a program combines a portion of the screen, a bit mask and the image itself using AND and XOR operations. This allows background graphics to appear in the areas of the rectangle outside of the actual image ("transparent bitmaps" in Microsoft lingo). For this to work, the bit mask must be a duplicate of the original, except that the image is totally black on a white background. The image bitmap must be enclosed in black. This can be done in code at run time, but it is both more convenient and faster to use predrawn images, usually as BITMAPs in the resource file. The animated image in MOVBMP (see Listing One, page 50, and Listing Two, page 51) is an iron cross with three layers of color on each arm and a mostly transparent center. Figure 1 shows the six bitmaps that are used. Background graphics will appear in the white areas of the mask. The top, right, bottom, and left bitmaps are used for rotating the animation sample. The "true image" bitmap shows the iron cross as it appears on screen.



In any graphics system, this type of animation is accomplished by saving the display area where the image will appear, ANDing the mask with the screen, and XORing the image with the ANDed result. New positions are then calculated, the saved image is restored, and the process is repeated. Depending on the bitmap size, the preceding steps may be enough. Windows is no speed demon, however, and display memory is relatively slow, so most programs require the use of memory device contexts (DCs). Memory DCs created with *CreateCompatibleDC* can be treated exactly like a display or other DC, depending on which type of DC is passed as an argument to this function. For larger bitmaps, additional efficiency measures are necessary.

Joe Sam is the owner of Autumn Software, which provides software and services for the AS/400, DOS, and Windows. He has been involved in consulting and commercial software for over 15 years and currently specializes in C, C++, and RPG. He can be contacted at P.O. Box 261646, Tampa, FL 33685.

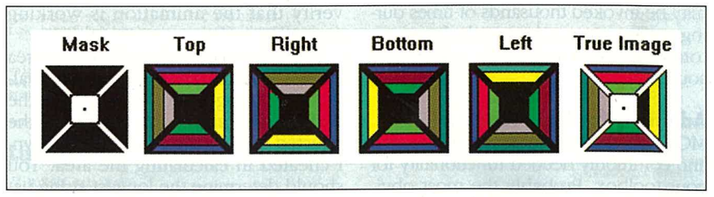


Figure 1: *Bitmaps used in the MOVBMP program. The animated image is an iron cross with three layers of color on each arm. The center is transparent.*

The Trouble with *BitBlt*

The *BitBlt* function provides a means to quickly get and put predrawn screen images. Unfortunately, when copying a displayed image to memory, *BitBlt* is more interested in what appears on the screen, and not necessarily what appears in your window. This means that if another window overlaps your client area, or even if one of your own menus is displayed over the animation, *BitBlt* copies the image from the overlapping display or menu. When your window regains focus, you may suddenly have a piece of another program's screen embedded in your window, which is generally unacceptable in a professional application. Also, due to the low priority and imprecision of timer messages, which are used to determine when to move the images, this state of events occurs more frequently than you might expect. This is a known problem that has occasionally popped up in question-and-answer columns. Microsoft is surely aware of the problem, but who knows when, or if, they will fix it, and, more importantly, if your users will have the fix. Fortunately, *BitBlt*s to the screen are confined to a window's client area.

An Optimization that Creates Work

The second feature to overcome involves a Windows 3.1 optimization. When *BeginPaint* is called, a structure is returned that contains, among other things, what Windows considers the invalid portion of your client area expressed as a rectangle. For the example in Figure 2, your program will get coordinates corresponding to rect A, C, G, F. However, Windows appears to clip to the region F, D, E, B, C; and the area A, B, E, D will not be repainted. This is good for Windows since less work and less time is necessary, but bad for you under certain circumstances. There's no fast, reliable method to determine if another window covers the image or if Windows will clean up the screen for you. In the end, you're pretty much on your own since most animations operate outside of the paint routine.

And there you have it: *BitBlt* may end up saving and restoring pieces of a menu or another program's display on your client area. The paint routine would seem to be a good place to get a clean copy of the background to avoid that situation, but part or all of the image can be inside the invalid rectangle yet outside of the clip region. In that case, if you resave the screen area, your image gets embedded in the screen in interesting ways.

Getting a Clean Image

My solution appears in *DoEraseBkgnd* (see Listing Two) which is called in response to the WM_ERASEBKGND message. The bypass timer flag is set (reset in *DoPaint*) to temporarily stop the animation, and the window DC is obtained to avoid clipping problems. The currently saved image is then restored to the screen. At that point, the DC is released and *DefWindowProc* is called to perform normal processing. If the image was in the invalid region, the screen is cleaned up. If not, the proper background is restored. Either way, you're then able to get a certifiably clean background image in *DoPaint*. This may seem inefficient, but the available alternatives take more code and more time, and are still imprecise (rectangles vs. regions again).

To automate the animation, a timer is used to send messages at approximately correct intervals and to allow the animation to proceed even when other windows have the focus. *GetNewTimer* illustrates obtaining a timer and allows the user, via a message box, to retry or cancel on failure (courtesy Charles Petzold).

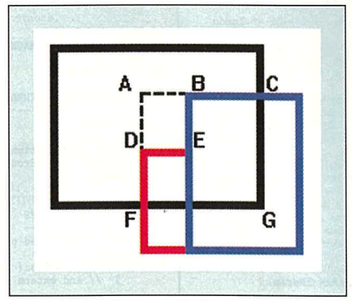


Figure 2: *The invalid portion of your client area expressed as a rectangle. Given the coordinates corresponding to rect A, C, G, F, Windows clips to the region F, D, E, B, C, and the area A, B, E, D is not repainted.*

Now that you can ensure clean images and know when to display them, how can you do so smoothly? Figure 3 illustrates a typical situation: The image is moving to the southeast. Rectangle 1 has just been displayed. Rectangle 2 will be displayed after restoring the screen. There is a large area of overlap and, if possible, you'd prefer to paint this area only once. This is actually what happens in MOVBMP. When the animation begins, a clean image of the screen is saved in the global *HBITMAP hbmOrg* (see *SelectBmp*) and maintained throughout the animation. The *Animate* routine handles all save, display, and restore operations using two additional bitmaps, the window DC, and two memory DCs.

In *Animate*, the old positions are saved and new ones calculated. *hbmOrg* is copied to *hbmSave*. The old and new positions are passed to *DiffORects*, which returns the client coordinates of the incremental area to be covered by the next display (the L-shaped area marked "2" in Figure 3) as two rectangles. These clean areas are copied from the screen to the proper positions in *hbmOrg*, as is the overlap portion from *hbmSave*. *hbmOrg* now has a complete clean image of the area to be used by the next display. *hbmOrg* is then copied to *hbmDest*. The bit mask is ANDed with *hbmDest* and the selected image is XORed with *hbmDest*. *BitBlt* accomplishes these tasks using the raster operation (ROP) codes SRCAND and SRCINVERT instead of the usual SRCCOPY. At this point, the new "transparent" image is complete. *DiffORects* is called again to determine the area to be restored. The same two rectangles are used, but the arguments are reversed, since we are now concerned

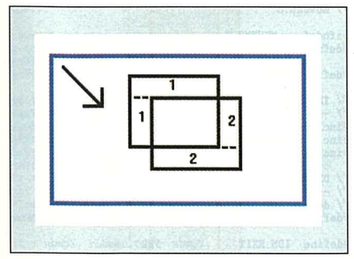


Figure 3: *Typical situation where the image is moving to the southeast, rectangle 1 has just been displayed, and rectangle 2 will be displayed after restoring the screen.*

with the L-shaped area marked "1" in Figure 3. With the values for these two differential rectangles in hand, the restore area is *BitBlt*ed to the screen from *hbmSave* and the new image is put to the screen from *hbmDest*. While there is a lot going on here, the work is done with very fast memory bitmaps and DCs. No redundant data is written to the display. *DiffORects* is the mainstay of this differential approach. The routine was developed empirically and has been (very) thoroughly tested. Because it works with client coordinates, the returned values are immediately usable. For smaller images, routines like *DiffORects* can be eliminated, but after about 20×20 pixels, flash and jerkiness become increasingly noticeable. The iron-cross images are 66×66 pixels.

Unless the Single Image menu option has been chosen, a different image is selected every other cycle. The bitmaps were drawn and loaded as top, right, bottom, and left images. By displaying the images in order, the illusion of rotation is generated in addition to the diagonal movement of the image. These could have been different colors, sunbursts, gradual fills, and so on, and are intended to demonstrate that animation does not have to involve moving to another area of the display. MOVBMP supports zero displacements so you can see pure rotation and even what appears to be a single stationary image.

Lastly, *Animate* cleans up the working DCs and bitmaps. Pay close attention to the reselection of the original or old bitmaps before deleting the DCs and work bitmaps. If this is omitted or done incorrectly, Windows will experience severe technical difficulties, as may your reputation, because the resulting loss in system resources continues for the duration of the session. Animation routines may be invoked thousands of times during a program run, so any leakage can totally deplete available system resources.

Additions for Your Toolbox

MOVBMP also illustrates both unusual and commonly needed functionality for your toolbox. Probably the most obvious is the elimination of the *switch* statement in *WndProc* via an array of message id/function pointer pairs (courtesy Ray Duncan). This is efficient for programmers because, among other benefits, new messages are easy to add, each message is related to a function (which helps avoid side effects and increases modularity), size boundaries can be calculated for you at compile time, and the basic skeleton can be used in every program. An extra function call is generated, but in practice this has been a small price to pay. Priority messages or those requiring special operations can be handled outside the array loop, as are WM_TIMER and WM_DESTROY. The message-associated functions take the standard *hwnd*, *msg*, *wParam*, and *lParam* arguments, which are *#defined* as STDWINARGS in the header file for use in function declarations and definitions. I've defined STDWINARGS in a base header that is included in every program.

Illustrations of font common-dialog usage are available, but little has been written about nondefault font initialization, as shown in Listing Two. Probably the best way to determine appropriate values is to write a small program using the font common dialog and use your debugger to view the returned LOGFONT structure. The font common dialog is also demonstrated in MOVBMP and used as a device to allow the background to be changed in order to verify that the animation is working properly.

Partial invalidation of the client area is used when displacement or timer values are changed. Although all of the code in *DoPaint* is invoked, only the affected area of the screen is redrawn. I cheated in calculating the area. You should determine the longest string using *GetTextExtent* for actual sizes.

Conclusion

I've tried to avoid "exercises for the reader," but any program can be improved. You might prefer to use a larger bitmap and one *BitBlt* to blast both the restore area and new image in one operation. I chose lesser memory usage and fewer calculations, and on my 386 20-MHz test machine, that is good enough. Modifying *SelectBmp* to allow loading any bitmap would be a more rewarding improvement. MOVBMP could then be used as a tool to determine optimal timer and displacement values for a given animation. The remainder of the program is not tied to specific bitmaps and may be used as is. MOVBMP is written as a single C-code module for simplicity; however, the animation functionality could easily be placed in a DLL or encapsulated in an *animator* class. The complete project, including resource, definition, and executable files is available electronically; see "Availability," page 2.

References

Duncan, Ray. "The Hazards of Exploring Evolving Environments." *PC Magazine* (April 13, 1993).

Petzold, Charles. *Programming Windows*. Redmond, WA: Microsoft Press, 1990.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 16.**

Listing One *(Text begins on page 48.)*

```
// movbmp.h

#ifndef    MOVBMP_H
#define    MOVBMP_H

#define STRICT

// INCLUDES
// --------
#include  <windows.h>
#include  <windowsx.h>
#include  <commdlg.h>

// DEFINES
// -------
// define standard windows arguments
#define STDWINARGS HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam

#define  IDM_EXIT            501

#define  IDM_FONT            505
#define  IDM_SELECTBMP       510
#define  IDM_UNSELECTBMP     511
#define  IDM_DDEC            515
#define  IDM_DINC            520
#define  IDM_TDEC            525
#define  IDM_TINC            530
#define  IDM_SINGLEIMAGE     535
// Timer ID
#define IMAGE_TIMER           1

// VARIABLE/USER-DEFINED TYPES
// ---------------------------
typedef struct
 { UINT   msgid;          // message id
 // pointer to a funtion taking std winproc parms
   LONG PASCAL (*pfnProc)( STDWINARGS );
 } WINMSGFN;


// FUNCTION PROTOTYPES
// -------------------
// Window and Exported procedures
extern "C" {
LRESULT WINAPI WndProc( STDWINARGS );
        }  // end extern statement

// message function procs
LONG PASCAL DoCmd( STDWINARGS );
LONG PASCAL DoEraseBkgnd( STDWINARGS );
LONG PASCAL DoPaint     ( STDWINARGS );
LONG PASCAL DoSize      ( STDWINARGS );

// Command message function procs
LONG PASCAL ChangeDisplacement( STDWINARGS );
LONG PASCAL ChangeTimer       ( STDWINARGS );
LONG PASCAL DoFont            ( STDWINARGS );
LONG PASCAL SelectBmp         ( STDWINARGS );
LONG PASCAL SingleImage       ( STDWINARGS );
LONG PASCAL UnSelectBmp       ( STDWINARGS );

// other functions
void Animate(HWND hwnd);
void DiffORects(RECT *rBmp, RECT *rRst, RECT *rUpper, RECT *rLower);
BOOL GetNewTimer(HWND hwnd);

#endif  /* movbmp.h */
```

Listing Two

```
// MOVBMP Main Module

#include "movbmp.h"

// pgm globals --------------------------
WINMSGFN msgfn[] =
     {
        WM_COMMAND,       DoCmd,
        WM_ERASEBKGND,    DoEraseBkgnd,
        WM_PAINT,         DoPaint,
        WM_SIZE,          DoSize,
     };
WINMSGFN cmdmsgfn[] =
     {
             IDM_DDEC,          ChangeDisplacement,
             IDM_DINC,          ChangeDisplacement,
             IDM_TDEC,          ChangeTimer,
             IDM_TINC,          ChangeTimer,
             IDM_FONT,          DoFont,
             IDM_SELECTBMP,     SelectBmp,
             IDM_SINGLEIMAGE,   SingleImage,
             IDM_UNSELECTBMP,   UnSelectBmp,
     };

// keep number of msgfn and cmdmsgfn elements
int  imsgfndim = (sizeof(msgfn) / sizeof(msgfn[0]));
int  icmdmsgfndim = (sizeof(cmdmsgfn) / sizeof(cmdmsgfn[0]));

HINSTANCE  hInstGBL;   // this instance
COLORREF   crrgbColors = RGB(0,0,255); // blue
RECT       rWndGBL = {0, 0, 0, 0}, rWndInv = {5, 0, 255, 0};
HDC        hdcOrg;
HBITMAP    hbmOrg, hbmOldOrg;
HBITMAP    hbm[5]; // top, right, bottom, left, background

// init the font to TrueType Arial, 72 points
LOGFONT    lfFontSel = {    -96,  // Height
                              0,  // Width
                            250,  // Escapement
                              0,  // Orientation
                            400,  // Weight
                           '\0',  // Italic
                           '\0',  // Underline
                           '\0',  // StrikeOut
                           '\0',  // CharSet
                         '\x03',  // OutPrecision
                         '\x02',  // ClipPrecision
                         '\x01',  // Quality
                         '\x22',  // PitchAndFamily
                        "Arial"  // FaceName
                       };

// flag variables
BYTE    btrblFlag = 0; // top/right/bottom/left flag
BYTE    bBypassTimer   = 255;  // 255 indicates first pass
BYTE    bBmpIsSelected = 0, bSingleImage = 0;

char    szAppName[] = "Move Bitmap", szClassName[] = "MOVBMP";

int     ndxGBL;  // work index
int     imageWidth, imageHeight, nTimerValue = 50;
int     nXDsp = 1, nYDsp = 1; // displacement values
int     nbmpX = 1, nDeltaX = 1, nbmpY = 1, nDeltaY = 1;

// begin --------------------------------------
#pragma argsused
int pascal WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR  lpszCmdLine,  int cmdShow)
 { HWND     hWnd;
   MSG      msg;
   WNDCLASS wndclass;
   char     szString[] = "Move Bitmap is ending...";

   hInstGBL = hInstance;  // save instance info
   // handle class registration and window creation
   if( !hPrevInstance )
   { wndclass.lpszClassName = szClassName;
     wndclass.hInstance     = hInstance;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.lpszMenuName  = "Main";
     wndclass.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
     wndclass.style         = NULL;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;

     if(!RegisterClass( &wndclass ))  //error
     { MessageBox(NULL, "Can't create window class.", szString,
                        MB_ICONEXCLAMATION | MB_OK);
       return 1;
     }
   }
   hWnd = CreateWindow( szClassName,
              szAppName,
          WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
          CW_USEDEFAULT,
          0,
          CW_USEDEFAULT,
          CW_USEDEFAULT,
          NULL,
          NULL,
          hInstance,
          NULL
            );
   if(hWnd == NULL)
   { MessageBox(NULL, "Can't create window.", szString,
                      MB_ICONEXCLAMATION | MB_OK);
     return 1;
   }
   ShowWindow (hWnd, cmdShow);
   UpdateWindow(hWnd);

   // pgm loop
   while ( GetMessage(&msg, NULL, 0, 0) )
   {
     TranslateMessage (&msg);
     DispatchMessage  (&msg);
   }
   return 0;
 } // end WNDMAIN


LRESULT WINAPI WndProc( STDWINARGS )
 { // handle Windows messages
   if( msg == WM_TIMER ) // here for faster handling
   { // animate if appropriate timer;
     if(wParam == IMAGE_TIMER && !bBypassTimer)  Animate(hwnd);
     return(0L);
   }
   for(ndxGBL = 0; ndxGBL < imsgfndim; ndxGBL++)
   { // walk messages array
     if(msg == msgfn[ndxGBL].msgid)  // if match, call Fn
        return ((*msgfn[ndxGBL].pfnProc) (hwnd, msg, wParam, lParam));
   }
   if( msg == WM_DESTROY ) // if not in array
   { // happens once; don't waste searches or fn
     if(bBmpIsSelected)  // clean up if bitmap was moving
        UnSelectBmp(hwnd, msg, wParam, lParam);
     PostQuitMessage (0);
     return(0L);
   }
   else return ( DefWindowProc(hwnd, msg, wParam, lParam) );
 }  // end WndProc


void Animate(HWND hwnd)  // Handle Bitmap Movement
 { static int  ncounter = 0; // when to inc btrblFlag
   int      nToX = 0, nToY = 0, nFromX = 0, nFromY = 0,
            nPrevX = nbmpX, nPrevY = nbmpY;
   RECT     rBmp, rRst, rUpper, rLower;
   HDC      hdcClient, hdcDest, hdcSave;
   HBITMAP hbmDest, hbmOldDest, hbmSave, hbmOldSave;

   hdcClient  = GetDC( hwnd);
   hdcDest    = CreateCompatibleDC( hdcClient);
   hdcSave    = CreateCompatibleDC( hdcClient);

   hbmSave = CreateCompatibleBitmap(hdcClient,
                                    imageWidth, imageHeight);
   hbmOldSave = (HBITMAP)SelectObject(hdcSave, hbmSave);

   // create destination bitmap
   hbmDest = CreateCompatibleBitmap(hdcClient,
                                    imageWidth, imageHeight);
   hbmOldDest = (HBITMAP)SelectObject(hdcDest, hbmDest);

   // determine border/direction changes, if any
   if( nbmpX < 1)  nDeltaX = nXDsp;
   else if( nbmpX > (rWndGBL.right-imageWidth) )
           nDeltaX = -(nXDsp);

   if( nbmpY < 1)  nDeltaY = nYDsp;
   else if( nbmpY > (rWndGBL.bottom-imageHeight) )
           nDeltaY = -(nYDsp);

   nbmpX += nDeltaX;  nbmpY += nDeltaY; // get new positions

   // copy clean image of saved screen
   BitBlt(hdcSave, 0, 0, imageWidth, imageHeight,
          hdcOrg,  0, 0, SRCCOPY);

   // set up incremental new screen image
   rBmp.left   = nbmpX;  rBmp.top = nbmpY;
   rBmp.right  = nbmpX + imageWidth  - 1;
   rBmp.bottom = nbmpY + imageHeight - 1;

   rRst.left   = nPrevX;  rRst.top = nPrevY;
   rRst.right  = nPrevX + imageWidth  - 1;
   rRst.bottom = nPrevY + imageHeight - 1;
   // get overlapping rect differentials
   DiffORects(&rRst, &rBmp, &rUpper, &rLower);
   // get clean incremental rects from screen
   BitBlt(hdcOrg,  rUpper.left - nbmpX, 0,
                  (rUpper.right - rUpper.left + 1),
                  (rUpper.bottom - rUpper.top + 1),
          hdcClient, rUpper.left, rUpper.top, SRCCOPY);

   BitBlt(hdcOrg,  rLower.left - nbmpX, rLower.top - nbmpY,
                  (rLower.right - rLower.left + 1),
                  (rLower.bottom - rLower.top + 1),
          hdcClient, rLower.left, rLower.top, SRCCOPY);

   // copy overlapping portion from previously saved screen image
```

(continued on page 52)

Listing Two *(Listing continued, text begins on page 48.)*

```
if(nbmpX < nPrevX)  nToX   = nXDsp; // if moving left
else                nFromX = nXDsp;

if(nbmpY < nPrevY)  nToY   = nYDsp; // if moving up
else                nFromY = nYDsp;

BitBlt(hdcOrg, nToX, nToY,
       imageWidth - nXDsp, imageHeight - nYDsp,
       hdcSave, nFromX, nFromY, SRCCOPY);

// set up initial destination image
BitBlt(hdcDest, 0, 0, imageWidth, imageHeight,
       hdcOrg, 0, 0, SRCCOPY);
// set up mask reusing hdcSave
SelectObject(hdcSave, hbm[4]);
// AND the mask
BitBlt(hdcDest, 0, 0, imageWidth, imageHeight,
       hdcSave,  0, 0, SRCAND);

// set up to put the bitmap
SelectObject(hdcSave, hbm[btrblFlag]);
// XOR the bitmap
BitBlt(hdcDest,  0, 0, imageWidth, imageHeight,
       hdcSave, 0, 0, SRCINVERT);

// restore incremental rects from old screen image
// use same rects with reverse order in fn call
DiffORects(&rBmp, &rRst, &rUpper, &rLower);
// put clean rects to screen
SelectObject(hdcSave, hbmSave);  // reselect hbmSave
BitBlt(hdcClient, rUpper.left, rUpper.top,
                  (rUpper.right  - rUpper.left + 1),
                  (rUpper.bottom - rUpper.top  + 1),
       hdcSave,   rUpper.left - nPrevX, 0, SRCCOPY);

BitBlt(hdcClient, rLower.left, rLower.top,
                  (rLower.right  - rLower.left + 1),
                  (rLower.bottom - rLower.top  + 1),
       hdcSave,   rLower.left - nPrevX,
                  (rLower.top -  nPrevY), SRCCOPY);

// PUT image to the screen
BitBlt(hdcClient, nbmpX, nbmpY, imageWidth, imageHeight,
       hdcDest, 0, 0, SRCCOPY);

if(!bSingleImage)
{ // change top, right, bottom, left every other display
  if(++ncounter >= 2)
  { // set bitmap index
    btrblFlag++;  ncounter = 0;
    if(btrblFlag > 3)  btrblFlag = 0; // reset flag/index
  }
}
// clean up memory DC's, work bitmaps
   SelectObject(hdcDest, hbmOldDest);
   DeleteDC(hdcDest);
   DeleteObject(hbmDest);

SelectObject( hdcSave, hbmOldSave);
DeleteDC( hdcSave);
DeleteObject( hbmSave );

ReleaseDC( hwnd, hdcClient);
} // end Animate


#pragma argsused
LONG PASCAL ChangeDisplacement( STDWINARGS )
 { // inc/dec x & y displacement by 1 to min/max
   if(wParam == IDM_DDEC) // decrement rqs
   {
     if(nXDsp < 1 || nYDsp < 1) // 0 pix min
        nXDsp = nYDsp = 1;
     else  nXDsp -= 1, nYDsp -= 1;
   }
   else // must have been increment rqs
   {
     if(nXDsp > 49 || nYDsp > 49) // 50 pix max
        nXDsp = nYDsp = 50;
     else  nXDsp += 1, nYDsp += 1;
   }
   // reset delta values
   nDeltaX = (nDeltaX < 1) ? -nXDsp : nXDsp;
   nDeltaY = (nDeltaY < 1) ? -nYDsp : nYDsp;

   InvalidateRect(hwnd, &rWndInv, TRUE); // redraw new values
   return (0L);
 } // end ChangeDisplacement


#pragma argsused
LONG PASCAL ChangeTimer( STDWINARGS )
 { // inc or dec timer interval by 50
   if(bBmpIsSelected)
   {
     KillTimer(hwnd, IMAGE_TIMER);
     bBypassTimer = 1; // set bypass flag
   }
   if(wParam == IDM_TDEC) // decrement rqs
   { // 50 ms min
     if(nTimerValue < 100)  nTimerValue  = 50;
     else                   nTimerValue -= 50;
   }
   else // must have been increment rqs
   { // 1000 ms max
     if(nTimerValue > 950)  nTimerValue  = 1000;
     else                   nTimerValue += 50;
   }
   InvalidateRect(hwnd, &rWndInv, TRUE); // redraw new values
   if(bBmpIsSelected)
   {
     if( !GetNewTimer(hwnd) )  // if can't get timer, end movement
          UnSelectBmp(hwnd, msg, wParam, lParam);
     bBypassTimer = 0; // reset bypass flag
   }
   return (0L);
 } // end ChangeTimer


void DiffORects(RECT *rBmp, RECT *rRst, RECT *rUpper, RECT *rLower)
 { // finds 2 rects portion of rRST outside overlap in
   // client coords, returns rects in rUpper and rLower
   rUpper->top = rRst->top; rLower->bottom = rRst->bottom;  // always

   if(rBmp->top > rRst->top)
     rUpper->bottom = rBmp->top - 1, rLower->top = rBmp->top;
   else
     rUpper->bottom = rBmp->bottom, rLower->top = rBmp->bottom + 1;

   if(rBmp->left > rRst->left)
   {
     rUpper->left = rLower->left = rRst->left;
     if(rBmp->top > rRst->top)
       rUpper->right = rRst->right, rLower->right = rBmp->left - 1;
     else  // rBmp->top < rRst->top
       rUpper->right = rBmp->left - 1, rLower->right = rRst->right;
   }
   else  // rBmp->left is same or less than rRst->left
   {
     rUpper->right = rLower->right = rRst->right;
     if(rBmp->top > rRst->top)
       rUpper->left = rRst->left, rLower->left = rBmp->right + 1;
     else
       rUpper->left = rBmp->right + 1, rLower->left = rRst->left;
   }
   return;
 } // end DiffORects


#pragma argsused
LONG PASCAL DoCmd( STDWINARGS )
 { // Handle WM_COMMAND msg, check wParam contents
   for(ndxGBL = 0; ndxGBL < icmdmsgfndim; ndxGBL++)
   {
     if(wParam == cmdmsgfn[ndxGBL].msgid)  // if match, call Fn
        return ((*cmdmsgfn[ndxGBL].pfnProc)(hwnd, msg, wParam, lParam));
   }
   if( wParam == IDM_EXIT ) // if not in array
   { // happens once; don't waste searches or fn
     SendMessage(hwnd, WM_CLOSE, 0, 0L);
     return(0L);
   }
   else return ( DefWindowProc(hwnd, msg, wParam, lParam) );
 } // end DoCmd


#pragma argsused
LONG PASCAL  // Handle WM_ERASEBKGND message
 DoEraseBkgnd( STDWINARGS )
 { // chk bypass - SOMETIMES get this msg before AND during
   // WM_PAINT/BeginPaint. Set flag here, DoPaint clears it.
   if(bBmpIsSelected && !bBypassTimer)
   { HDC  hdc;
     // WGO-- BitBlt captures image from overlapping screen
     // when bmp is underneath. So work around...  If this
     // restore is erroneous, ERASE gets it in invalidated
     // region. If valid, the restoration remains.
     // Done here due to timing and because BeginPaint
     // CLIPS to the invalid rectangle (or smaller)
     // so our work would be for nothing in that routine.
     bBypassTimer = 1; // set bypass timer routine flag.
     hdc = GetDC( hwnd );  // restore saved screen
     BitBlt(hdc, nbmpX, nbmpY, imageWidth, imageHeight,
            hdcOrg, 0, 0, SRCCOPY);
     ReleaseDC( hwnd, hdc);
   }
   return ( DefWindowProc(hwnd, msg, wParam, lParam) );
 } // end DoEraseBkgnd


 #pragma argsused
LONG PASCAL  // Handle IDM_FONT Menu message
 DoFont( STDWINARGS )
 { CHOOSEFONT cfStruct;
   LOGFONT    lfStruct = lfFontSel;

   cfStruct.lStructSize = sizeof(CHOOSEFONT);
   cfStruct.hwndOwner   = hwnd;
   cfStruct.lpLogFont   = &lfStruct;
   cfStruct.Flags       = CF_SCREENFONTS | CF_EFFECTS |
                          CF_INITTOLOGFONTSTRUCT;
   cfStruct.rgbColors   = crrgbColors;

   if( ChooseFont(&cfStruct) )
   {
     lfFontSel = lfStruct;
     lfFontSel.lfEscapement = 250;
     crrgbColors = cfStruct.rgbColors; // set selected color
     // redraw with selected font
     InvalidateRect(hwnd, &rWndGBL, TRUE);
   }
   return (0L);
 }  // end DoFont
```



```
#pragma argsused
LONG PASCAL   // Handle WM_PAINT message
 DoPaint( STDWINARGS )
 { HDC         hdc;
   HFONT       hFont = NULL, hFontPrev = NULL;
   PAINTSTRUCT ps;
   char        szOutString[40];
   int         nlength;

   hdc = BeginPaint(hwnd, &ps);
   // draw timer and displacement values
   nlength = wsprintf(szOutString, "Timer Value is %d.", nTimerValue);
   ExtTextOut(hdc, 5, rWndGBL.bottom - 60, 0, NULL, szOutString,
              nlength, NULL);

   nlength = wsprintf(szOutString, "X Displacement is %d.", nXDsp);
   ExtTextOut(hdc, 5, rWndGBL.bottom - 40, 0, NULL,
              szOutString, nlength, NULL);

   nlength = wsprintf(szOutString, "Y Displacement is %d.", nYDsp);
   ExtTextOut(hdc, 5, rWndGBL.bottom - 20, 0, NULL,
              szOutString, nlength, NULL);

   SetBkMode(hdc, TRANSPARENT);
   // create and select the font
   hFont     = CreateFontIndirect(&lfFontSel);
   hFontPrev = (HFONT)SelectObject(hdc, hFont);

   SetTextAlign(hdc, TA_BASELINE | TA_CENTER);
   // create shadow by drawing in black first, one over
   ExtTextOut(hdc, (rWndGBL.right >> 1) + 1,
                   (rWndGBL.bottom >> 1) + 1,
                   0, NULL, "U S A",  5, NULL);
   SetTextColor(hdc, crrgbColors); // now in selected color
   ExtTextOut(hdc, rWndGBL.right >> 1,
              (rWndGBL.bottom >> 1),
              0, NULL, "U S A",  5, NULL);

   if(hFontPrev) // delete the font
   { SelectObject(hdc, hFontPrev);
     DeleteObject(hFont);
   }
   if(bBmpIsSelected) // get fresh screen image
      BitBlt(hdcOrg, 0, 0, imageWidth, imageHeight,
             hdc, nbmpX, nbmpY, SRCCOPY);

   EndPaint( hwnd, &ps);
   bBypassTimer = 0; // reset bypass flag
   return(0L);
 } // end DoPaint


#pragma argsused
LONG PASCAL DoSize( STDWINARGS )
 { // Handle Wm_SIZE message
   if( rWndGBL.right  == LOWORD(lParam) &&
       rWndGBL.bottom == HIWORD(lParam) )
     return 0L;  // if same size, exit

   // otherwise get new size
   rWndGBL.right  = LOWORD(lParam);
   rWndGBL.bottom = HIWORD(lParam);
   // update invalid text rect
   rWndInv.top    = rWndGBL.bottom - 60;
   rWndInv.bottom = rWndGBL.bottom;

   if(bBypassTimer == 255)
   { // if first pass, exit after setting size
     bBypassTimer = 0;
     return 0L;
   }
   if(wParam == SIZE_MAXIMIZED || wParam == SIZE_RESTORED)
   { // redraw on new size
     if(bBmpIsSelected != 2)
        InvalidateRect(hwnd, &rWndGBL, TRUE);
     else // restart if was icon and movement sel
        SelectBmp(hwnd, msg, wParam, lParam);

     return 0L;
   }
   if(wParam == SIZE_MINIMIZED && bBmpIsSelected == 1)
   { // If going to icon, stop movement, set sel to 2 for restore.
     UnSelectBmp(hwnd, msg, wParam, lParam);
     bBmpIsSelected = 2;  // set Flag for size return
   }
   return (0L);
 }  // end DoSize


BOOL GetNewTimer(HWND hwnd)
 { // set up timer
   while( !SetTimer(hwnd, IMAGE_TIMER, nTimerValue, NULL) )
   {
     if( IDCANCEL == MessageBox( hwnd,
                     "Can't get timer for animation.",
                     "Move Bitmap Information",
                     MB_ICONEXCLAMATION | MB_RETRYCANCEL) )
          return (FALSE);
   }
   return (TRUE); // got timer
 } // end GetNewTimer


#pragma argsused
LONG PASCAL   // Handle IDM_SELECTBMP Menu message
 SelectBmp( STDWINARGS )
 { HDC     hdcClient;
   BITMAP bm;  // Bitmap info
   HMENU  hmenuMain = GetMenu(hwnd);

   if( !GetNewTimer(hwnd) )
        return 0L;  // return if can't obtain timer

   if(wParam == IDM_SELECTBMP) // chg menu select to stop
      ModifyMenu(hmenuMain, IDM_SELECTBMP, MF_BYCOMMAND,
                            IDM_UNSELECTBMP, "Stop Movement");
   // load bitmaps from resource file
   hbm[0] = LoadBitmap(hInstGBL, "image1"); // top
   hbm[1] = LoadBitmap(hInstGBL, "image2"); // right
   hbm[2] = LoadBitmap(hInstGBL, "image3"); // bottom
   hbm[3] = LoadBitmap(hInstGBL, "image4"); // left
   hbm[4] = LoadBitmap(hInstGBL, "imagebkg"); // mask

   GetObject(hbm[0], sizeof(bm), (LPSTR)&bm);
   imageWidth  = bm.bmWidth;  imageHeight = bm.bmHeight;
   btrblFlag   = 1;  // set top as first image

   hdcClient = GetDC( hwnd);
   hdcOrg  = CreateCompatibleDC( hdcClient);
   hbmOrg  = CreateCompatibleBitmap(hdcClient,
                                    imageWidth, imageHeight);
   hbmOldOrg = (HBITMAP)SelectObject(hdcOrg, hbmOrg);
   // save initial screen image
   BitBlt(hdcOrg, 0, 0, imageWidth, imageHeight,
          hdcClient, nbmpX, nbmpY, SRCCOPY);

   // clean up
   ReleaseDC(hwnd, hdcClient);
   bBmpIsSelected = 1;
   return (0L);
 } // end SelectBmp


#pragma argsused
LONG PASCAL   // Handle IDM_SINGLEIMAGE command
 SingleImage( STDWINARGS )
 { // Toggle Single Image/Multiple image switch
   int   nChkType;
   HMENU  hmenu = GetMenu(hwnd);

   bSingleImage = !bSingleImage;
   nChkType = (bSingleImage) ? MF_CHECKED : MF_UNCHECKED;
   CheckMenuItem(hmenu, IDM_SINGLEIMAGE, MF_BYCOMMAND | nChkType);
   return(0L);
 } // end SingleImage


#pragma argsused
LONG PASCAL   // Handle IDM_UNSELECTBMP
 UnSelectBmp( STDWINARGS )
 { HMENU hmenu = GetMenu(hwnd);
   HDC    hdcClient;

   KillTimer(hwnd, IMAGE_TIMER);
   bBmpIsSelected = 0;
   // restore the screen
   hdcClient = GetDC( hwnd);
   BitBlt(hdcClient, nbmpX, nbmpY, imageWidth, imageHeight,
          hdcOrg, 0, 0, SRCCOPY);
   ReleaseDC( hwnd, hdcClient);

   if(wParam == IDM_UNSELECTBMP)
   {  // if menu call, redraw: image may be under menu
     InvalidateRect(hwnd, &rWndGBL, TRUE);
     // chg menu select to start
     ModifyMenu(hmenu, IDM_UNSELECTBMP, MF_BYCOMMAND,
                       IDM_SELECTBMP, "Start Movement");
   }
   for(ndxGBL = 0; ndxGBL < 5; ndxGBL++) // delete the bitmaps
       DeleteObject( hbm[ndxGBL] );

   SelectObject(hdcOrg, hbmOldOrg);  // deselects hbmOrg
   DeleteDC(hdcOrg);
   DeleteObject(hbmOrg);

   return (0L);
 } // end UnSelectBmp
```

End Listings

Writing a Multimedia App in Liana

Dynamic behavior requires a dynamic language

Jack Krupansky

Three fashionable topics that get a lot of talk are multimedia, Windows, and object-oriented programming (OOP). But sometimes it's hard to find instances in which these technologies are applied to real-world problems. In this article, I'll talk about a multimedia application that's currently in day-to-day use at a number of corporations and describe how it was developed.

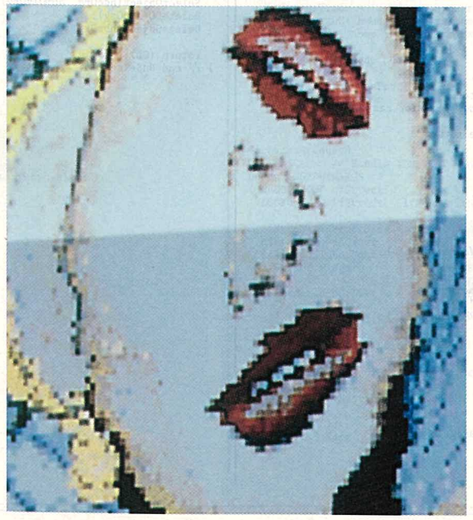
I developed Avenue as a consulting project for VIS Development Corp. (Waltham, MA). In implementing Avenue, I used the Liana object-oriented programming language and class library. Liana is a development tool I created that lets you build small to medium-size Windows applications very quickly. (See Ray Valdés's "The Liana Programming Language," *DDJ*, October 1993.)

Enhancing Training Videos

Although businesses have much money invested in training videos, managers may wonder if these videos are effective in imparting knowledge, given the passive nature of video-watching. To make the learning experience more interactive, VIS Development offers a service that digitizes existing training video tapes, places them on CD-ROM, and structures them via an interactive graphical front end: Avenue. The result is a package that is more engaging to users. The package can also gather statistics on usage, as well as conduct tests on comprehension.

Jack is president of Base Technology, where he develops and markets Liana and consults on development of Windows applications. He can be reached at 303-440-4558, by fax at 303-444-4186, on CompuServe at 70642,2662, or at 1320 Pearl St., Suite 110-B, Boulder, CO 80302.

Last year, VIS staff prototyped Avenue using a DOS-based multimedia authoring system. Then they decided to move it to Windows using Asymetrix ToolBook. However, progress soon stalled because the staff lacked programming experience. When I was brought on board to implement the application, I proposed Liana as a way of obtaining a quality result as soon as possible. Liana's interpretive nature and high level of abstraction free you from much of the details of C/C++ programming and the Windows API. Although Liana did not have any multimedia support, a quick review of the Media Control Interface (MCI) led me to the conclusion that a Liana *video* class would be very easy to implement. I was able to get a preliminary version of the class running within a few hours. But before discussing my Liana implementation, I'll describe Avenue further.



Overview of Avenue

Avenue allows the user to walk up to a PC and quickly select and start a video training course. The hierarchical structure of the front end allows the user to visually navigate to the desired training video; see Figure 1.

Avenue is really a meta-application, akin to an authoring tool. It adds menus, which can be full-screen bitmap images, and buttons, which can be of arbitrary size. It operates via keyboard or mouse, but is also well suited for use with touch screens. The images may be as simple as text menus or as sophisticated as scanned photographs or diagrams that allow users to visually select an item of interest. A selected item might immediately bring up a training-video segment, or, alternately, a more-detailed image might be shown so that the user can make a more precise selection. The author of the training application can also specify an initial logon dialog, so that the material can be tailored to each user's skill level. Progress through the video can also be a result of passing tests in previous training sessions.

Course authors, managers, and system administrators can customize a particular training video via a set of scripts and configuration files. In addition to controlling the order of presentation of video segments, the scripts define the hierarchical structure of the presentation and can fine-tune the appearance of the screen.

The users of Avenue-produced training materials vary widely in skill level, depending on the course material; they can include factory workers and sales personnel, as well as corporate managers. Some users are comfortable with a dense set of choices and controls; others will want a friendlier, sparser presentation of choices with fewer controls. Avenue can accommodate changes without requiring expensive video editing or custom software development.

Avenue has a facility for logging who watched what, and for how long. It also has a testing capability with features that can be used to survey users, as well as test for comprehension.

Books and Chapters

In Avenue parlance, each of the top-level menus is known as a "book," while an individual training video is a "chapter." The user selects a chapter from a book; a chapter is, of course, organized as a sequence of pages. Each page consists of a video clip and optional text and graphics.

The chapter's author decides whether a given page should auto-advance to the next page upon completion of that segment or whether to give the user complete control. A standard complement of VCR-like buttons is available, including previous page, rewind, play/pause, forward, and next page. The Pages button brings up a list box of all page titles so the user can randomly go to any page. The Index button brings up an alphabetical index for the chapter to also allow random selection. The author can decide to share an index between multiple chapters; the index supports branching across chapters. The Mark button allows the user to set bookmarks and return to them later. The More button allows the user to request more detail in the form of a nested chapter-within-a-chapter.

The course author can associate a test with a given chapter. A test is really just a specialized chapter in which each question is a page. A test page often has a full-screen bitmap that displays the question and multiple-choice answers using a mix of fonts, colors, and graphics. A test question may also be posed by a video clip associated with the page. Another video clip can be shown if the user gets an incorrect answer. The Avenue script file offers the course author a lot of flexibility in designing the test and in how to respond to user responses.

Avenue supports various ways of logging user activity. These include recording what chapters have been viewed, which pages have been seen, the time spent in each chapter or page, and the results of any tests taken. An instructor can find out not just a user's test score, but also how each question was answered, in how many attempts. There is more to Avenue than I have the space for, but the description here should give you an idea of the implementation task I faced.

Application Code

Due to space limits and proprietary considerations, I'll show only an abridged version of Avenue's source code. Hopefully, the excerpts in Listings One through Five will give you a good idea of what's involved in writing a nontrivial Liana application. There is additional code in the electronic version of this article; see "Availability," page 2.

The application consists of a main function (Listing One, page 56) and a number of classes. Similar to C, the Liana interpreter calls *main* after initializing the environment. Unlike C, a Liana main function initializes the application and then quickly returns. The Liana interpreter then waits until the application's windows have been closed before cleaning up and exiting. Liana automatically dispatches Windows events by calling member functions for the Liana object associated with each window.

The hierarchy of open books is stored in a Liana stack object created in the main function. The *stack* class (Listing Two, page 56) is derived from the Liana dynamic-array class and is a good example of how easy it is to manipulate objects in Liana. Since Liana is a typeless language, there's no need to have separate classes or templates for every type of object that you wish to store on a stack or in an array. The same class is used for the chapter stack.

The two primary classes in Avenue are *bookwin* (Listing Three, page 56), which manages a book window, and *chapwin* (Listing Four, page 56), which manages a chapter window. The *touch_menu* class (available electronically) is common to both *bookwin* and *chapwin* and manages the display of a bitmap and the selection of soft buttons. A Liana *displaylist* is used to represent the soft buttons. Other classes in Avenue include a *book_entry* class, which is based on the Liana *figure* class, plus a user (skill) level, a path string, and a label string. There is also a *figure* class, which represents the rectangular area to be defined for each soft button used to bring up books. Avenue's *touch_menu* class is a general-purpose class that could be reused in other applications.

The *chapfile* class manages a single chapter file, which consists of a text file with a bunch of header lines followed by one line per page. Each page line is stored in a page object; the page objects are stored in a Liana dynamic ar-

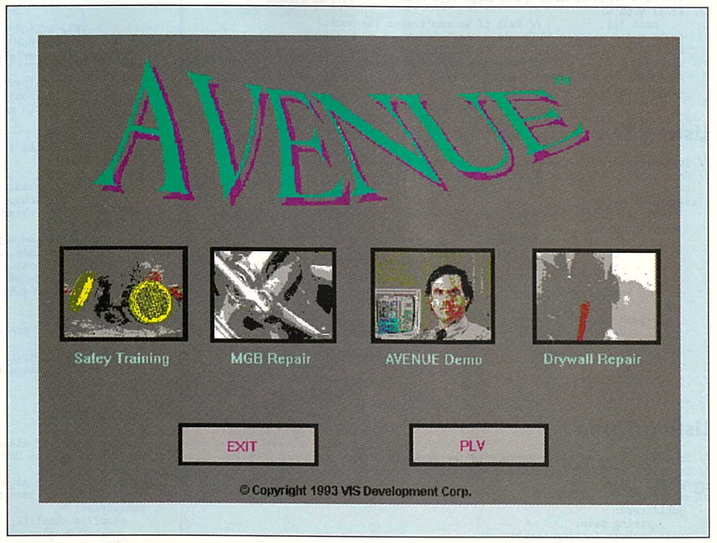


Figure 1: *The Avenue interface.*

ray. The Liana *range* class is used to store the range of video frames for a chapter page. It is another good example of the flexibility of Liana, since the start and end of the range can be of any type: integer, real, string, or a user-defined class.

Avenue's *chapter_state* class is used to record the state of a chapter, including the path of the chapter file, the information about the current page of the chapter, the current video-frame number, and whether video is actually playing. The *index_entry* class is based on the *chapter_state* class because an index entry needs much of the same information, plus the text to be displayed in the index. Avenue supports index entries that branch across chapters and even to a specific frame within a page.

Finally, the Liana *video* class (Listing Five, page 57) encapsulates the Windows Media Control Interface (MCI) API. This class also provides useful features such as the option of displaying a bitmap over the video window. At the end of Listing Five, you can see how a DLL entry point is declared in Liana. The Avenue implementation does not use the *video* class directly, but instead derives class *chvideo* from it. This is for several reasons:

- To detect frame transitions so Avenue can update bitmaps when the end of a chapter page is reached.
- To customize error handling.
- To get control when the video finishes playing a range of frames.
- To get control whenever a seek is performed on the video so the bitmaps can be synchronized in case the page number changes. The *frame*, *mci_error*, *succeeded*, and *position_set* member functions are called for those events, so they are redefined by the *chvideo* class.

Project Assessment

Unlike many commercial projects, the implementation effort on Avenue was rather modest. As the the only developer, I started the project in late September 1992, working less than half-time. The first product shipment occurred barely four months later in early February 1993.

Avenue consists of about 3000 lines of Liana code. Based on my experience implementing Windows apps in C using the Windows SDK (Liana is one such instance), I estimate that a native C implementation of Avenue would require about 30,000 lines of code and would have taken perhaps ten times longer. Estimating a C++ implementation is a bit trickier, given the various class libraries and application frameworks, but I suspect it would require 15,000 lines of C++ and take perhaps five times longer.

Would an application generator have helped? Not really. Avenue's user interface is dynamic in nature and is not something that can be statically defined with a dialog editor, resource toolkit, or window builder.

The only real downside to using Liana is that it's a proprietary language from a one-person company (however, full source code is available). Also, some programmers prefer a more comprehensive interactive development environment (IDE). Even so, Liana's prototyping speed merits your consideration.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**

Listing One *(Text begins on page 54.)*

```
// Main entry point for Avenue --- by Jack Krupansky.
// Copyright 1993 VIS Development Corporation

void main ()
{   // Access the private profile (.INI) file for AVENUE.
    env = new envfile (path (getcwd (), apname, ".ini"));

    // Determine whether status line is enabled for books.
    if ((string s = env ["enable_book_status_line"]) == "")
        enable_book_status_line = true;
    else
        enable_book_status_line = bool (int (s));

    // Create initial book window.
    window w = new bookwin (path (argv [1], apname));
    if (! w.path)
        exit ();            // Exit if we can't open the book.
    books = new stack;  // Create a stack for the book hierarchy.
    book << w;              // Put the initial book on the stack.
    w.show;                 // Make the book window visible.
    // main() then returns to Liana to wait until application is
    // terminated by the user.
}
```

Listing Two

```
// Implementation of class Stack  --- by Jack Krupansky
// Copyright 1993 John W. Krupansky d/b/a Base Technology

class stack: array {
     bool isempty () {  return this.size == 0;   }
     any pop () {
          if (int i = this.size) {
            any v = this [i - 1];
            trim (1);
          }
          return v;
     }
     any push (any v) { this << v;  return v;   }
     void stack () { array (); } // constructor calls base class
     any top ()  {
         if (int i = this.size)   return this [i - 1];
         else                     return null;
     }
};
```

Listing Three

```
// Implementation of class Bookwin --- by Jack Krupansky
// Copyright 1993 VIS Development Corporation

class bookwin: touch_menu {
    publicread:
        string path;
    void bookwin (string path);
    void clicked (int i)  {
        if (i == -1)
            this.status = "Nothing selected, try again...";
        else {
            string path = this [i].path;
            if (path == "*exit")        exit_book ();
            else                        new_chapter (i);
        }
    }
    bool close ();
    void destroy ();
    void exit_book ();
    void new_chapter (int course)  {
        any c = this [course];
        if (! access (string path = c.path)) {
            warn ("Unable to access map/chapter file: "+path);
            this.status = "Select something...";
            return;
        }
        this.status = "Loading "+c.text; // Let user know what we're doing
        this.cursor = "wait";            // Display the hour-glass cursor.
        window cur_map = books.top;      // Get top book from the stack.
                                         // See if we're going to another
        if (ext (path) == ".CHP") {      // book or to a chapter.
            if (chapwin      // See if we're going to the same chapter.
                && path == chapwin.path) {

                this.status = "Resuming chapter...";
            } else {
                if (! chapwin)   chapwin = new chapwin;
            }
            chapwin.path = path;          // Open the new chapter.
            if (chapwin.path.size)
                chapwin.show_maximized;  // Make the chapter visible.
        } else {
            window new_map = new bookwin (path);    // Open a new book window.
            if (new_map.path) {
                books << new_map;        // Push the book on the stack.
                new_map.show;            // Make the new book window visible
                this.hide;               // Hide the current book.
            }
        }
        this.cursor = "normal";          // Restore the cursor.
    }
    void path_set (string path);
    int user_level_set (int user_level);
};
```

Listing Four

```
// Implementation of class Chapwin -- by Jack Krupansky.
// Copyright 1993 VIS Development Corporation

class chapwin: touch_menu {
      chapter_state incorrect_video_saved_state;
   publicread:
      chapfile chapfile;
      pushbutton play_button;
      chvideo tv;
      string path;
      int page;
      page p;
      int correct_answers;
public:
   bool page_ended;

   void chapwin () {
        touch_menu ();              // Initialize object for parent class.
        chapters = new stack;       // Create stack for nested chapters.
                                    // Create the buttons at top of window.
        this
           << (prev_page_button = new pushbutton ("<<-", "prev_page"))
           << (play_button = new pushbutton ("Play","toggle_play_video",8))
            // ...more buttons...;
   }
   void chose (int i);
   void clear_touch_area_definitions ();
   void clicked (int i);
   void close ();
   answer_dialog get_answer_dialog (string code);
   void force_store_playing_time (bool finished);
   void go_forward ();

   void load_touch_area_definitions (string path);
   void moved (x, y);
   void next_page () {this.page++;}
   int page_finished ();
   int page_set (int page);
   void paint (x1, y1, x2, y2);
   void paint_status ();
   void paint_msg_status ();
   void paint_page_status ();
   void paint_title_status ();
   string path_set (string path);
   void pause_video ()  {
        if (tv.playing) {
            tv.pause;                           // Stop the video.
                                                // Toggle label of Play btn
            play_button.text = "Play";          // from Pause back to Play.
                                                // Display status for user.
            this.status = "Pausing video at frame "+tv.position;
            force_store_playing_time (false);  // Record user statistics.
        }
   }
   range play_range ();
   void play_video ()  {
        if (! tv.playing) toggle_play_video ();
   }
   void position (x, y);
   int position_set (int position);
   void prev_page () {this.page--;}
   void push_to_chapter ();
   void play_video ();
   void refresh ();
   void return_from_chapter ();
   void rewind_video ();
   void set_bookmark ();
   void set_button_visibility (pushbutton button, bool visible);
   void set_control_visibility (control c, bool visible);
   void show_help ();
   void show_index ();
   void show_more_detail ();
   void show_pages ();
   chapter_state state ();
   chapter_state state_set (chapter_state s);
   string status_set (string status);
   bool status_line_enabled_set (bool v) {return v;}
   void store_playing_time (bool finished);
   void toggle_play_video ()  {
        if (tv.playing)   pause_video ();
        else {
            if (page_ended)          // If user presses play at end of a page
                this.page++;         // Then go on to the next page.
            tv.play;                 // Start the video playing.
            started_play = time (); // Record the time spent playing video.
            play_button.text = "Pause"; // Change label of Play btn to Pause
            this.status = "Playing frames "      // Display status for user.
                          + tv.frames.start

                          + " to " + tv.frames.end;
        }
   }
   void video_finished ();
};
```

Listing Five

```
// Implementation of class Video, which encapsulates the Windows
// Windows Media Control Interface (MCI)
// Copyright 1993 John W. Krupansky d/b/a Base Technology

class video: custombutton {
    static int vid_gen;
    string vid;
    timer frame_timer;
    range frames;
    bool video_visible;
 publicread:
    string path;
    bitmap bitmap;
    bitmap bitmap_set (bitmap bitmap);
    int clean_position (int pos)  {
        return max (1, min (this.size, pos));
    }
    void close ();
    void destroy ();
    void frame (int frame) {}
    range frames () {return frames.copy;}
    range frames_set (range r);
    long handle_message (
        unsigned hWnd,
        unsigned message,
        unsigned wParam,
        long lParam);
    bool mci_execute (string command);
    void mci_error (string command, string msg);
    void show_mci_error (string command, string msg);
    bool mci_execute (string command, int hWnd);
    void paint (int x1, int y1, int x2, int y2) {paint ();}

    void paint ();
    string path_set (string path);
    void pause ()  {
        if (path && ! ready () && ! stopped ()) {
            mci_execute ("pause "+vid);
            frames.start = this.position;
        }
    }
    void play () {
        pause ();
        if (path) {
            if (frames.start != null)   play (frames);
            else {
                mci_execute ("play "+vid+" notify", handle);
                this.video_visible = true;
                start_timer ();
            }
        }
    }
    void play (range range) {play (range.start, range.end);}
    void play (int start, int end) {
        if (path) {
            this.frames.range (start, end);
        pause ();
        start = clean_position (start);
        end = clean_position (end);
        string cmd = "play "+vid;
        if (this.position != start)
          cmd += " from "+start;
        cmd += " to "+end+" notify";
        mci_execute (cmd, handle);
        this.video_visible = true;
        start_timer ();
      }
    }
    bool playing () {return this.status == "playing";}
    int position () {return int (status ("position"));}
    int position_set (int position)
    {
      position = clean_position (position);
      if (this.position != position) {
        if (path) {
          if (bool not_ready = ! ready ())
            pause ();
          mci_execute ("seek "+vid+" to "+position);
          frames.start = position;
          if (not_ready)
            play ();
        }
      } else
        frames.start = position;
      return position;
    }
    bool ready () {return status ("ready") == "true";}
    void rewind ();

    int size () {return int (status ("length"));}
    void start_timer ();
    string status () {return status ("mode");}
    string status (string item)
    {
      if (path) {
        string result = "";
        memory buf = malloc (256);
        string command = "status "+vid+" "+item;
                                          // call the MCI DLL entry points
        if (int rv = mciSendString (command, buf, buf.size, handle)) {
            mciGetErrorString (rv, buf, buf.size);  // another DLL entry point
            warn ("MCI Error", command+"\n\n"+buf);
        } else {
            result = ""+buf;
        }
        free (buf);
        return result;
      }
    }
    void step ();
    void stop ();
    bool stopped () {return status ("mode") == "stopped";}
    void succeeded ();
    void video ();
    void video (string path);
    bool video_visible () {return status ("video") == "on";}
    bool video_visible_set (bool video_visible);
    };
unsigned long mciSendString (
    char *lpstrCommand,
    char *lpstrReturnString,
    unsigned wReturnLength,
    unsigned hCallback) "mmsystem.dll:mciSendString"
bool mciGetErrorString (
    unsigned long wError,
    char *lpstrBuffer,
    unsigned uLength) "mmsystem.dll:mciGetErrorString"
```

End Listings

The VESA BIOS Extension/Audio Interface

A standard software interface for audio

Doug Cody

The Video Electronics Standards Association (VESA, for short), is an international, nonprofit standards organization responsible for creating the Super-VGA BIOS standards, that has developed and published the VESA BIOS Extension/Audio Interface (VBE/AI) for AT-class PCs. The main goal of VBE/AI is to provide a standard software interface for audio, similar to the ROM BIOS for video. Just as the lack of Super-VGA standards has hindered application development in the video world, the lack of sound standards has limited the sound-board industry. At the current time, support for individual audio boards falls on the shoulders of third-party developers. In addressing this void, VBE/AI has three main goals:

- Provide a hardware-independent approach to controlling digital (WAVE) audio, MIDI, and volume control.
- Hide implementation details, so the code can be implemented as a ROM BIOS, DOS device driver, or TSR.
- Give the interface comparable functionality with Windows, NT, and OS/2, so these OSs can support the interface through virtual services.

The following article briefly describes various aspects of the VBE Audio Interface.

Doug is a software engineer at MediaVision and a member of the VBE/AI committee. He can be contacted at MediaVision, 47300 Bayside Parkway, Fremont, CA 94538.

VESA Audio Services

The video BIOS in PCs provides a set of functions to set video modes, read/write characters to the screen, scroll the screen, and so on. Figure 1 shows the original services. The VESA Super-VGA BIOS extensions started higher numerically to coexist with the standard BIOS. In the first release, the extensions appeared similar to the standard set; see Figure 2. In the next couple of releases, a provision was made to allow extensions for other types of devices, starting at function number 0x4F10. All new VESA standard devices will be allocated a particular ID, as in the case of the Audio Interface, which was assigned 0x13. Figure 3 lists the complete set of VBE/AI Int 10H functions. The primary purpose of the Int 10 services is to allow the application to query, open, and close the device. Access to the device API itself is performed through 32-bit far calls using the Pascal calling convention.

Since there can be multiple audio devices in one system, VBE/AI allows for multiple software APIs, one per device. A handle gives the application access to one device API. To use a device, the application locates a device handle, queries the specific device via the handle, then opens and closes the device via the handle.

Before jumping into specific device services, I just want to skim over the query process. Within VBE/AI, each device provides two data structures. One of these contains information about the device, and the other contains the specific entry points of the device services. The query process allows the application to retrieve a copy of the information structure. This structure contains information such as hardware ID, configuration, and capabilities. The format of this structure is not yet complete at the time of this writing, but it is worth mentioning because it is an integral part of the VBE/AI interface.

WAVE Audio Interface

The Wave device class supports two common metaphors for handling the data: single block and continuous streaming. The single-block operation performs the operation on the data block, issues an interrupt when done, then returns to an idle state. The streaming mode attempts to keep data moving to and from the hardware and doesn't stop until explicitly told to by the application. In this mode, completion interrupts are presented to the application at fixed points during the streams I/O. These approaches model the real-world hardware operation of the 8237 DMA controller, of which there are two in every AT-class machine. Most audio cards make use of a DMA channel; some use two. Other devices, such as the Turtle Beach Multisound card, make use of memory mapping with on-board memory, using the CPU to move the data instead of a DMA channel. Since only the metaphor of single-block and continuous modes are characterized in the interface, these non-DMA devices can easily be supported.

When a VBE/AI Wave device is opened, the application is given a copy of its services structure. Listing One, page 62, illustrates the Wave Services structure. The data fields in the structure are for identification purposes, or reserved for future updates of the VBE/AI release. The remaining portion of the services structure presents subroutine calls for the application and two 32-bit *far* callbacks for the application to fill out. These callbacks allow the VBE/AI code to call the application on certain events.

The services can be lumped into three groups: housekeeping, process control, and block I/O. Each of these services is described in Figure 4. Note *wsTimerTick*. In the VBE/AI definition, the application "owns" the system-clock timer. The VBE/AI device specifies how many callbacks per second are needed to keep running. The application is responsible for calling this subroutine at the requested rate.

The two callbacks from the device to the application provide the application with real-time notification that certain events have occurred. These two callbacks indicate the completion of block I/O performed by the device. There is one callback for completion of PCM playback, and one for PCM record; see Figure 5. The two-callback approach was adopted so that full duplex (that is, simultaneous play and record) could be performed using the VBE Audio Interface.

When a VBE/AI device is opened, the application is given a copy of its services structure

MIDI Interface

The MIDI device class in its simplest form uses the metaphor of a MIDI transmitter for playing notes. This is similar to the Windows approach that views all musical devices, such as an FM chip, wave-synthesis card, or MIDI port, as a common device type. The VBE/AI MIDI device is defined as a general-MIDI device, so its channel and patch (instrument) assignments are consistent with the International MIDI Association's definition of general MIDI. Unlike Windows, though, there is no differentiation between low- and high-end synthesizers. This scheme, within Windows, was implemented by allocating channels 1–10 for the high-end synthesizer, and channels 11–16 for a low-end synthesizer. In the general-MIDI specification (and therefore in the VBE/AI interface), only channel 10 is recognized as a special channel. All others are free to be used any way the application desires. (Incidentally,

Function	Description
AH=00	Set mode
AH=01	Set cursor type
AH=02	Set cursor position
AH=03	Get set cursor position
AH=04	Get dight pen position
AH=05	Set display page
AH=06	Initialize or scroll up window
AH=07	Initialize or scroll down window
AH=08	Read character and attribute at cursor
AH=09	Write character and attribute at cursor
AH=0A	Write character at cursor
AH=0B	Set palette, background, or border
AH=0C	Write graphics pixel
AH=0D	Read graphics pixel
AH=0E	Write character in teletype mode
AH=0F	Get video mode

Figure 1: *Standard Int 10h functions.*

AH=4F VESA subfunctions.
AL=00 VBE 1.0—GET Super-VGA information
AL=01 VBE 1.0—GET Super-VGA mode information
AL=02 VBE 1.0—SET Super-VGA video mode
AL=03 VBE 1.0—GET Current video mode
AL=04 VBE 1.0—Save/Restore Super-VGA video state
AL=05 VBE 1.0—CPU video memory control
AL=06 VBE 1.1—Get/Set logical scan-line length
AL=07 VBE 1.1—Get/Set display start
AL=08 VBE 1.2—Get/Set DAC palette control

Figure 2: *VESA Int 10h extensions.*

Function	Description
AH=4F	VESA Audio Interface subfunctions
AL=13	VBE/AI interface ID
BL=00	Check VBE/AI presence
BL=01	Locate a device
BL=02	Query a device
BL=03	Open the device
BL=04	Close the device
BL=05	Reserved
BL=06	Reserved
BL=07	Reserved

Figure 3: *VESA Int 10h extensions for the audio interface.*

channel 10 in the general-MIDI specification is the percussive-instrument channel.)

When a VBE/AI MIDI device is opened, a copy of its services structure is given to the application; Listing Two, page 62, illustrates this structure. The structure, like the wave-services structure, contains few data variables, several subroutine calls, and a couple of callbacks to the application.

The only data variable presently worth looking at is the *mspatches* bit field. This array of 256 bits indicates which general-MIDI patches are currently preloaded, and thus available for use. For simple MIDI transmitters and receivers, this bit field could have all bits set, since it can only "assume" that the external device contains the required patches. For devices like an FM synthesizer that have no memory, this table may have all bits set to 0, thus indicating it will need patch data from the application before the instrument is played. Later on, I'll discuss the VBE/AI unique approach to loading patch data into the device.

Again, the services can be lumped into three groups: housekeeping, patch handling, and MIDI I/O; see Figure 6. Additionally, there are two callbacks from the device to the application to provide the application with real-time notification that certain events have occurred. The first callback allows downloaded patch memory blocks to be returned to the application. The second callback lets VBE/AI MIDI receivers send inbound MIDI data to the application in real time; see Figure 7.

Function	Description
(a)	
wsDeviceCheck	Retrieve run-time and miscellaneous information from the device.
wsGetLastError	Retrieve the last error posted by the device.
wsTimerTick	The application is responsible for calling this subroutine at the rate requested by the VBE/AI device.
(b)	
wsPlayBlock	Play a single block of PCM data, then return to idle.
wsPlayCont	Start and continue playing PCM from a circular buffer, giving callbacks at fixed points. Application is responsible for handling the circular buffer.
wsRecordBlock	Record a single block of PCM data, then return to idle.
wsRecordCont	Start and continue playing PCM from a circular buffer, giving callbacks at fixed points. The application is responsible for handling the circular buffer.
(c)	
wsPCMInfo	Receive information on the upcoming PCM data stream: the PCM data size, sample rate, compression type, and stereo or mono.
wsPauseIO	Pause the data stream.
wsResumeIO	Resume a paused state, created by *wsPauseIO*.
wsStopIO	Immediately stop any I/O and return the device to an idle state.

Figure 4: *Wave-device services: (a) housekeeping, (b) block I/O; (c) process control.*

DOS and Software Standards for Audio

At the Game Developer's Conference held earlier this year in Santa Clara, California, Tom Rettig of Broderbund Software held a round-table discussion on the sound issues facing software developers. A list of topics was fielded from the audience and rated by importance. By an overwhelming majority, the number-one topic was the need for a DOS audio software standard.

You may be wondering why there is such concern when the world seems to be moving to platforms like Windows and OS/2 that already support an audio interface. Indications are that the games industry is leaning towards staying in DOS via 32-bit DOS extenders. DOS is still the optimum game environment due to having direct control over the machine. So, if the games don't move soon to these new OSs, PC-audio industry revenue will be tied to DOS for quite a while. There's still no compelling business audio application that creates revenue like the DOS entertainment market.

It may seem ironic that games developers are asking for a software interface, yet want to have direct control over the hardware. In fact, most DOS software developers demand, as an inalienable right, the knowledge to directly program the hardware. Up until 1990, software vendors had few audio-hardware selections, so support was a relatively easy endeavor. For music, there was the Adlib card and Roland MPU-401. For voice, there was Creative Labs' SoundBlaster and Covox Speech Thing. But when all else failed, the lowly PC speaker could still be used to play tones and rudimentary sounds.

Since 1990, an explosion of audio hardware with varying architectures has entered the PC sound market. Each software company that has maintained in-house support for sound has had to carry the burden of adding support for these new cards. As a result, third-party software libraries for audio became very popular in 1991 as the number of cards grew.

But due to the plethora of architectures, the industry is swaying under the burden of trying to support too much with too little resources. There are well over a dozen hardware interfaces a developer will need to support, in order to support all PC audio-board products. The showing of hands at the Game Developer's Conference suggests that software developers are ready to take one step back off the hardware, but first need a uniform approach to programming audio. The trade-off of using a hardware-independent software interface, vs. creating, maintaining, and growing a complete library, is now a prudent choice of engineering time.

Not everyone has come to this conclusion, however. A few die-hard game developers have retreated to a position that they will only support a few audio cards. Unfortunately, this doesn't serve the customer, who might have an unsupported audio board, nor does it foster development of advanced technologies. From a competitive standpoint, these developers will ultimately suffer, since competitors will gain a technical one up by taking advantage of the new whiz-bang features found on the newer cards. The old lesson is, standing still only allows your competitors to pass you by.

—D.C.

One issue addressed by the VBE/AI interface is that most PC MIDI hardware doesn't contain full general-MIDI patch sets. For devices such as the OPL2 FM chip (as found on the original Adlib FM card), there's no memory within the chip to hold all the general-MIDI instruments. So, either the driver takes up an exorbitant amount of memory to cache the instrument bank, or the application will have to provide the patches from an external source, such as a disk-resident library. The latter approach of storing the patches to reside on disk has been adopted for the VBE audio interface. To do this, the architecture has been divided in two parts. The first part is the hardware-specific driver to translate the MIDI stream into audible sound, as described earlier. The second part is an optional disk-resident, vendor-specific library that holds the individual instrument settings, known as *patches*.

The role of the patch library is to allow the application to present a given patch to the hardware before the instrument is played. To integrate hardware dependencies in this approach, the application need only "understand" the skeletal structure of the library. In other words, the app must know how to get to the patch data, but doesn't have to understand the contents of the patch. When a program change is required, the application will search the library, extract the patch, and pass this to the device using the *msPreLoadPatch* subroutine. For patches larger than available memory, a protocol has been defined for sending patch data in smaller chunks.

One benefit to this architecture is that applications can substitute their own patches for the ones in the vendor's library. There's no rule saying the patches *must* come from the vendor's own library. Of course, this does mean the application is delving into hardware dependencies.

Volume Control

The VBE/AI Volume services is an API for controlling both volume and mixer channels. Within the API are setting controls for stereo volume, sound positioning within a 3-D space using a spherical-coordinate system, parametric equalizer, and filter control. The 3-D sound positioning, which is derived from the virtual-reality field, is the most exciting aspect of this interface. In new products coming to market this year, 3-D audio will become an important selling feature. Two-dimensional schemes such as Qsound and Surround Sound can also work with this interface.

The approach to the volume/mixer class is different than the MIDI and Wave device classes. When opened, the MIDI and Wave devices are considered owned until closed, whereas a volume device can be acquired by any number of applications, and is thus not exclusively

Function	Description
wsApplPSyncCB	Performed when playback has completed for either a single-shot block or a division of the circular buffer.
wsApplRSyncCB	Performed when recording has completed for either a single-shot block or a division of the circular buffer.

Figure 5: *Callback functions.*

Function	Description
(a)	
msDeviceCheck	Retrieve run-time and miscellaneous information from the device.
msGetLastError	Retrieve the last error posted by the device.
msGlobalReset	Reset/silence any active voices, free up all allocated voice to channel assignments, and so on. MIDI transmitter/receivers will send the All Notes Off message.
msTimerTick	In the VBE/AI definition, the application owns the timer. The VBE/AI device specifies how many callbacks per second are needed to keep running. The application is responsible for calling this subroutine at the requested rate.
(b)	
msMIDImsg	Take the MIDI data stream and interpret it to create the appropriate sound on the device. The data stream is defined in the MIDI specification.
msPollMIDI	On most PC MIDI transmitter/receiver devices, hardware interrupts are used to receive the inbound MIDI data. For those hardware devices that use only a polling process, this routine allows the application to receive the waiting MIDI data. All inbound data is sent to the application via an application-supplied callback routine, described in Figure 7.
(c)	
msPreLoadPatch	This is the first of two subroutines for handling downloading of patch data from the application. The application passes a memory pointer to the patch to the device. The device can retain the pointer as long as needed, or until the application requests it to be released.
msUnloadPatch	This is the second of two subroutines for handling downloading of patch data from the application. Using this function, the application tells the device to release the memory associated with a downloaded patch.

Figure 6: *MIDI services: (a) Housekeeping; (b) for MIDI I/O; (c) patch handling.*

Function	Description
msApplFreeCB	Used by the VBE/AI device to return any patch block received via the *msPreLoadPatch* subroutine.
msApplMIDIIn	Used by MIDI receivers to send captured inbound MIDI data to the application.

Figure 7: *MIDI callback functions.*

owned. This way, the user may have volume control through such things as a TSR, even if the application does not provide one.

An interesting aspect is that volume service APIs are acquired without performing an Int 10 *Device Open* function these services are acquired through the functions rather than the device open functions.

When a VBE/AI Volume device is acquired, a copy of its services structure is given to the application; see Listing Three, page 62. The structure (like the wave-services structure) contains few data variables and several subroutine calls. The services can be lumped into two groups: housekeeping and device control; see Figure 8.

Function	Description
(a)	
vsDeviceCheck	Retrieve run-time and miscellaneous information from the device.
vsGetLastError	Retrieve the last error posted by the device.
(b)	
vsSetVolume	Define two levels for volume control. A user setting from 0.0 to 1.0 and the true volume setting. The user setting is unique in that it provides the user with a control, independent from the application's use. This allows the user to adjust the setting to his/her environment.
vsSetFieldVol	Provide 3-D sound positioning. If the device supports 2-D (beyond normal left/right stereo panning) such as Qsound or Surround Sound.
vsToneControl	Tone control is a programmable interface for parametric equalizers.
vsFilterControl	Filter control gives control of a high-pass and/or low-pass filter for the given channel.
vsOutputPath	On some mixer or volume devices that support multiple outputs, this function allows the output to be switched.
vsResetChannel	Allows the application to request the volume or mixer channel to be set to its default state.

Figure 8: *Volume and mixer services: (a) housekeeping; (b) device control.*

Next Up

In future releases of the VBE/AI specification, you can expect to see a 32-bit audio interface and sound-effects processing control. In addition, VESA has active working committees developing open standards in the areas of video interfacing (VAVI), feature connector (VAFC), and multimedia data channels (VM-Channel).

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 9.**

Listing One

```
typedef struct {

    // housekeeping

    char wsname[4];      // name of the structure
    long wslength;       // structure length

    char wsfuture[16];       // 16 bytes for future expansion

    // device driver functions

    long (pascal far *wsDeviceCheck ) (int, long );
    int  (pascal far *wsPCMInfo     ) (int, long, int, int );
    int  (pascal far *wsPlayBlock   ) (void far *, long );
    int  (pascal far *wsPlayCont    ) (void far *, long, long);
    int  (pascal far *wsRecordBlock ) ();
    int  (pascal far *wsRecordCont  ) (void far *, long, long);
    int  (pascal far *wsPauseIO     ) ();
    int  (pascal far *wsResumeIO    ) ();
    int  (pascal far *wsStopIO      ) ();
    void (pascal far *wsTimerTick   ) ();
    int  (pascal far *wsGetLastError) ();

    // callback filled in by the application

    void (pascal far *wsApplPSyncCB ) (int, void far *, long ); //play
    void (pascal far *wsApplRSyncCB ) (int, void far *, long ); //rec

} WAVEService, far *fpWAVServ;
```

Listing Two

```
typedef struct {

    // housekeeping

    char msname[4];         // name of the structure
    long mslength;      // structure length

    // runtime data

    int  mspatches[16];     // patches loaded table bit field
    char msfuture[16];      // 16 bytes for future expansion

    // device driver functions

    long ( pascal far *msDeviceCheck ) (int, long );
    int  ( pascal far *msGlobalReset ) ();
    int  ( pascal far *msMIDImsg     ) (char far *, int );
    int  ( pascal far *msPollMIDI    ) (int );
    int  ( pascal far *msPreLoadPatch) (int, int, void far *, long );
    int  ( pascal far *msUnloadPatch ) (int, int );
    void ( pascal far *msTimerTick   ) ();
    int  ( pascal far *msGetLastError) ();

    // callbacks filled in by the application

    void ( pascal far *msApplFreeCB ) ( int, int, void far * );
    void ( pascal far *msApplMIDIIn ) ( int, int, char );

} MIDIServ, far *fpMIDServ;
```

Listing Three

```
typedef struct {

    // housekeeping

    char vsname[4];      // name of the structure
    long vslength;       // structure length

    char vsfuture[16];       // 16 bytes for future expansion

    long (pascal far *vsDeviceCheck  ) (int , long );
    void (pascal far *vsSetVolume    ) (int , int , int );
    long (pascal far *vsSetFieldVol  ) (int , int , int );
    int  (pascal far *vsToneControl  ) (int , int , int );
    long (pascal far *vsFilterControl) (long );
    void (pascal far *vsOutputPath   ) (int );
    void (pascal far *vsResetChannel ) ();
    int  (pascal far *vsGetLastError ) ();

} VolumeService, far *fpVolServ;
```

End Listings

DSP and Audio Compression

Enhancing audio compression performance

Jay B. Reimer

By its very nature, audio compression is a real-time process. You simply cannot afford to have gaps during the processing of data that lead to dropouts in the output audio. Historically, the architecture of the PC's operating environment—that is, its operating system and hardware—wasn't designed to optimally support real-time operation. With increasing demands placed on the PC's host processor to support graphics and provide access to file systems, audio processing is relegated to whatever MIPs are left over. This means that as the system designer, you have two choices: shoehorn the audio processing into the leftover MIPS, or have a dedicated coprocessor that can meet the real-time requirements of audio processing.

This is where the benefits of digital-signal processors (DSPs) become apparent. The architecture of DSPs has, from its inception, been optimized to support the needs of real-time systems. The DSP is designed to do a great deal of work in each clock cycle. This unit of work is consistent and includes the ability to perform a multiply-accumulate calculation (MAC) in the same time the processor can perform an add calculation or other ALU operation. Many microprocessors still require multiple cycles for add and ALU operations, although RISCs only require one. Since the essence of many of the audio-compression techniques (LPC, ADPCM, MPEG) is a "sum-of-products" type calculation, DSPs provide perfect support for them. This article explores the various audio-compression techniques and how DSPs can be used to improve performance.

Jay, a senior member of the technical staff at Texas Instruments, has been involved in every generation of TMS320 digital-signal processors. He has also been involved in the development of a number of audio and communications algorithms and systems. Jay can be contacted at Texas Instruments, P.O. Box 1443, M/S 701, Houston, TX 77251-1443.

Audio-compression Techniques

To reconstruct an analog signal, basic sampling theory dictates that the sample rate be based on the signal's bandwidth. For audio signals, the sampling frequency for selected inputs is shown in Figure 1. The second major attribute of a sampled signal is the resolution, or number of bits, that are used to digitize the input waveform. Most PC sound systems offer 8 bits of resolution, while music or CD-quality systems require 16 bits of resolution. These two attributes—sampling rate and resolution—determine the overall data rate of the system and the total amount of memory required to store a digitized signal.

A basic technique to capture digitized audio and preserve it is called linear pulse code modulation (PCM). PCM samples an audio signal at a given frequency with a selected resolution to generate a digital representation of an audio input. PCM requires a large amount of memory to store a digital representation of a sampled signal. One minute of uncompressed, stereo, CD-quality audio can consume close to ten Mbytes of storage.

Therefore, one of the biggest challenges associated with the use of audio information is the implementation of efficient compression algorithms that minimize the amount of storage space required for a sampled signal. By selecting the most effective audio-compression technique, you can minimize the amount of memory required to store a sampled audio signal while producing an output audio signal of acceptable quality. Some of the more popular audio-compression techniques are shown in Figure 2.

PRODUCT DEBUT

Just look at you,

reading this space and there's not even an ad in it.

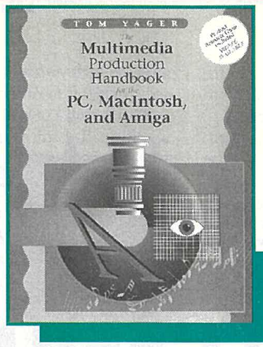
What if this space held important developer information?

You might heed advice, act on an offer, even spend money.

In fact, there are **115,000 professional developers** who could be persuaded to do the same.

All because of one little ad!

Call Cheryl Canion at *Dr. Dobb's Journal*
(415-358-0126 x306)
if you'd like to tap into this powerful place of persuasion.

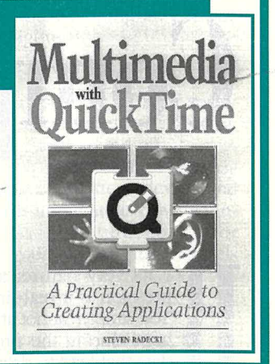


The First State-of-the-Art Multiplatform, Hands-on Guide

The Multimedia Production Handbook for the PC, Macintosh, and Amiga
Tom Yager

Now Available!

Paperback: $35.95, 382 pp.
ISBN: 0-12-768030-6



Create Your Own QuickTime... Color-Sound-Animation-Video

Multimedia with QuickTime
Steven Radecki

Now Available!

Paperback: $32.95, 350 pp.
ISBN: 0-12-574750-0

For our Free Catalog: **e-mail:** acad.com
Or write: AP PROFESSIONAL, 525 B Street, San Diego, CA 92101-4495

Order from your local bookseller *or directly from*

AP PROFESSIONAL
Order Fulfillment Dept. **DM**17915
6277 Sea Harbor Drive, Orlando, FL 32887

CALL TOLL FREE
1-800-321-5068
FAX **1-800-336-7377**

Prices subject to change without notice. © 1993 by Academic Press, Inc. All Rights Reserved. RYAN/LMW/ST —14123

CIRCLE NO. 1212 ON READER SERVICE CARD

It's FREE! Absolutely, positively, totally FREE! No strings! No commitment! No shipping! No handling! No nothing! FREE! FREE! FREE! FREE!

It's your FREE introductory issue of

Metatec's NautilusCD℠

The Multimedia Magazine℠ on CD ROM

This is not a misprint. It's an honest-to-gosh offer for a FREE introductory issue of NautilusCD, The Multimedia Magazine on CD ROM.

NautilusCD is the premier showcase of multimedia creativity. Our contributors are multimedia's superstars, wizards who can make your computer screen flash to life with moving pictures...play music or sing for you...dazzle your eyes with 3-D graphics...and even talk to you! It's an experience not to be missed. You'll get a subscription offer with your FREE issue, but no more will come – and neither will any bills – until you tell us "OK, I've gotta have it!" So call for your FREE issue now!

To order call toll-free 1-800-448-2323
Available in Macintosh and Windows versions. CD ROM drive required.
Limited Time Offer. One CD ROM per inquiry. International callers-1 614 766-3165

NautilusCD The Multimedia Magazine
7001 Discovery Boulevard, Dublin, Ohio 43017

CIRCLE NO. 1211 ON READER SERVICE CARD

STOP

Losing Out On Multimedia!

ADD VALUE TO YOUR PROJECTS WITH AWARD WINNING SOUND*

$19 to $39 New Wav/MCI **DIGITAL AUDIO DONGLES**
SoundJr Player Modules, Self-Powered, USA Made

- AUTO INSTALL / NO JUMPERS!
- NO BATTERIES
- USE ON LAPTOPS
- PROGRAMMER TOOLS $39 - $79
- USE FOR COPY PROTECTION

Silence

*Most Valuable Product '92 - PC Computing - Power Japanese

by SiliconSoft Ph: 408-446-4521 Fax: 408-446-5196
(Formerly Silicon Shack) 4760 Castlewood Drive, San Jose, CA 95129

Hot Line: 800-969-4411 Ask for FREE Product Info

™SoundJr, SiliconSoft are trademarks of SiliconSoft, Inc. Other product name are trademarks of their manufacturers.

CIRCLE NO. 1213 ON READER SERVICE CARD

Dr. Dobb's JOURNAL SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

READER SERVICE CARD • FREE INFORMATION

CIRCLE 1000 FOR 1 YEAR SUBSCRIPTION TO *DR. DOBB'S JOURNAL* AND BE BILLED LATER FOR ONLY $25. (U.S. ONLY)
Please help us help you by taking the time to answer a few questions. Thank you very much!

Name _____
Title _____
Company _____
Address _____
City _____ State ____ Zip _____
Phone (____) _____

MM Supp.93: Use before April 30, 1994

A. Which (if any) of the following applications are you currently developing?
1. ☐ Database applications
2. ☐ Windows 3.X applications
3. ☐ UNIX applications
4. ☐ Embedded systems applications

B. Please describe your involvement in the decision to purchase:
5. ☐ Authorize
6. ☐ Recommend
7. ☐ Specify

C. When will you purchase the products you are inquiring about?
8. ☐ Purchase within one month
9. ☐ Purchase within six months
10. ☐ General Information

Use this card to find out about the products and services listed in this issue. Simply circle the appropriate numbers below.

Inquiry Numbers 1–490

1	2	3	4	5	6	7	8	9	10	11	12	13	14
15	16	17	18	19	20	21	22	23	24	25	26	27	28
29	30	31	32	33	34	35	36	37	38	39	40	41	42
43	44	45	46	47	48	49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80	81	82	83	84
85	86	87	88	89	90	91	92	93	94	95	96	97	98
99	100	101	102	103	104	105	106	107	108	109	110	111	112
113	114	115	116	117	118	119	120	121	122	123	124	125	126
127	128	129	130	131	132	133	134	135	136	137	138	139	140
141	142	143	144	145	146	147	148	149	150	151	152	153	154
155	156	157	158	159	160	161	162	163	164	165	166	167	168
169	170	171	172	173	174	175	176	177	178	179	180	181	182
183	184	185	186	187	188	189	190	191	192	193	194	195	196
197	198	199	200	201	202	203	204	205	206	207	208	209	210
211	212	213	214	215	216	217	218	219	220	221	222	223	224
225	226	227	228	229	230	231	232	233	234	235	236	237	238
239	240	241	242	243	244	245	246	247	248	249	250	251	252
253	254	255	256	257	258	259	260	261	262	263	264	265	266
267	268	269	270	271	272	273	274	275	276	277	278	279	280
281	282	283	284	285	286	287	288	289	290	291	292	293	294
295	296	297	298	299	300	301	302	303	304	305	306	307	308
309	310	311	312	313	314	315	316	317	318	319	320	321	322
323	324	325	326	327	328	329	330	331	332	333	334	335	336
337	338	339	340	341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360	361	362	363	364
365	366	367	368	369	370	371	372	373	374	375	376	377	378
379	380	381	382	383	384	385	386	387	388	389	390	391	392
393	394	395	396	397	398	399	400	401	402	403	404	405	406
407	408	409	410	411	412	413	414	415	416	417	418	419	420
421	422	423	424	425	426	427	428	429	430	431	432	433	434
435	436	437	438	439	440	441	442	443	444	445	446	447	448
449	450	451	452	453	454	455	456	457	458	459	460	461	462
463	464	465	466	467	468	469	470	471	472	473	474	475	476
477	478	479	480	481	482	483	484	485	486	487	488	489	490

Inquiry Numbers 491–980

491	492	493	494	495	496	497	498	499	500	501	502	503	504
505	506	507	508	509	510	511	512	513	514	515	516	517	518
519	520	521	522	523	524	525	526	527	528	529	530	531	532
533	534	535	536	537	538	539	540	541	542	543	544	545	546
547	548	549	550	551	552	553	554	555	556	557	558	559	560
561	562	563	564	565	566	567	568	569	570	571	572	573	574
575	576	577	578	579	580	581	582	583	584	585	586	587	588
589	590	591	592	593	594	595	596	597	598	599	600	601	602
603	604	605	606	607	608	609	610	611	612	613	614	615	616
617	618	619	620	621	622	623	624	625	626	627	628	629	630
631	632	633	634	635	636	637	638	639	640	641	642	643	644
645	646	647	648	649	650	651	652	653	654	655	656	657	658
659	660	661	662	663	664	665	666	667	668	669	670	671	672
673	674	675	676	677	678	679	680	681	682	683	684	685	686
687	688	689	690	691	692	693	694	695	696	697	698	699	700
701	702	703	704	705	706	707	708	709	710	711	712	713	714
715	716	717	718	719	720	721	722	723	724	725	726	727	728
729	730	731	732	733	734	735	736	737	738	739	740	741	742
743	744	745	746	747	748	749	750	751	752	753	754	755	756
757	758	759	760	761	762	763	764	765	766	767	768	769	770
771	772	773	774	775	776	777	778	779	780	781	782	783	784
785	786	787	788	789	790	791	792	793	794	795	796	797	798
799	800	801	802	803	804	805	806	807	808	809	810	811	812
813	814	815	816	817	818	819	820	821	822	823	824	825	826
827	828	829	830	831	832	833	834	835	836	837	838	839	840
841	842	843	844	845	846	847	848	849	850	851	852	853	854
855	856	857	858	859	860	861	862	863	864	865	866	867	868
869	870	871	872	873	874	875	876	877	878	879	880	881	882
883	884	885	886	887	888	889	890	891	892	893	894	895	896
897	898	899	900	901	902	903	904	905	906	907	908	909	910
911	912	913	914	915	916	917	918	919	920	921	922	923	924
925	926	927	928	929	930	931	932	933	934	935	936	937	938
939	940	941	942	943	944	945	946	947	948	949	950	951	952
953	954	955	956	957	958	959	960	961	962	963	964	965	966
967	968	969	970	971	972	973	974	975	976	977	978	979	980

Inquiry Numbers 981–1470

981	982	983	984	985	986	987	988	989	990	991	992	993	994
995	996	997	998	999	1000	1001	1002	1003	1004	1005	1006	1007	1008
1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022
1023	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036
1037	1038	1039	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050
1051	1052	1053	1054	1055	1056	1057	1058	1059	1060	1061	1062	1063	1064
1065	1066	1067	1068	1069	1070	1071	1072	1073	1074	1075	1076	1077	1078
1079	1080	1081	1082	1083	1084	1085	1086	1087	1088	1089	1090	1091	1092
1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103	1104	1105	1106
1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119	1120
1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134
1135	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148
1149	1150	1151	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162
1163	1164	1165	1166	1167	1168	1169	1170	1171	1172	1173	1174	1175	1176
1177	1178	1179	1180	1181	1182	1183	1184	1185	1186	1187	1188	1189	1190
1191	1192	1193	1194	1195	1196	1197	1198	1199	1200	1201	1202	1203	1204
1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215	1216	1217	1218
1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231	1232
1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246
1247	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260
1261	1262	1263	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274
1275	1276	1277	1278	1279	1280	1281	1282	1283	1284	1285	1286	1287	1288
1289	1290	1291	1292	1293	1294	1295	1296	1297	1298	1299	1300	1301	1302
1303	1304	1305	1306	1307	1308	1309	1310	1311	1312	1313	1314	1315	1316
1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327	1328	1329	1330
1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343	1344
1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358
1359	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372
1373	1374	1375	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386
1387	1388	1389	1390	1391	1392	1393	1394	1395	1396	1397	1398	1399	1400
1401	1402	1403	1404	1405	1406	1407	1408	1409	1410	1411	1412	1413	1414
1415	1416	1417	1418	1419	1420	1421	1422	1423	1424	1425	1426	1427	1428
1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439	1440	1441	1442
1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455	1456
1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT 740 PITTSFIELD, MA

POSTAGE WILL BE PAID BY ADDRESSEE

Dr. Dobb's JOURNAL SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

READER SERVICE MANAGEMENT DEPT.
P.O. BOX 5215
PITTSFIELD, MA 01203-9857

Dr. Dobb's JOURNAL SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

READER SERVICE CARD • FREE INFORMATION

CIRCLE 1000 FOR 1 YEAR SUBSCRIPTION TO *DR. DOBB'S JOURNAL* AND BE BILLED LATER FOR ONLY $25. (U.S. ONLY)
Please help us help you by taking the time to answer a few questions. Thank you very much!

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT 740 PITTSFIELD, MA
POSTAGE WILL BE PAID BY ADDRESSEE

Dr. Dobb's JOURNAL SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER
READER SERVICE MANAGEMENT DEPT.
P.O. BOX 5215
PITTSFIELD, MA 01203-9857

Name _____
Title _____
Company _____
Address _____
City _____ State ____ Zip _____
Phone (____) _____

A. Which (if any) of the following applications are you currently developing?
1. ❒ Database applications 2. ❒ Windows 3.X applications
3. ❒ UNIX applications 4. ❒ Embedded systems applications

B. Please describe your involvement in the decision to purchase:
5. ❒ Authorize 6. ❒ Recommend
7. ❒ Specify

C. When will you purchase the products you are inquiring about?
8. ❒ Purchase within one month 9. ❒ Purchase within six months
10. ❒ General Information

MM Supp.93: Use before
April 30, 1994

Use this card to find out about the products and services listed in this issue. Simply circle the appropriate numbers below.

Inquiry Numbers 1–490

1	2	3	4	5	6	7	8	9	10	11	12	13	14
15	16	17	18	19	20	21	22	23	24	25	26	27	28
29	30	31	32	33	34	35	36	37	38	39	40	41	42
43	44	45	46	47	48	49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80	81	82	83	84
85	86	87	88	89	90	91	92	93	94	95	96	97	98
99	100	101	102	103	104	105	106	107	108	109	110	111	112
113	114	115	116	117	118	119	120	121	122	123	124	125	126
127	128	129	130	131	132	133	134	135	136	137	138	139	140
141	142	143	144	145	146	147	148	149	150	151	152	153	154
155	156	157	158	159	160	161	162	163	164	165	166	167	168
169	170	171	172	173	174	175	176	177	178	179	180	181	182
183	184	185	186	187	188	189	190	191	192	193	194	195	196
197	198	199	200	201	202	203	204	205	206	207	208	209	210
211	212	213	214	215	216	217	218	219	220	221	222	223	224
225	226	227	228	229	230	231	232	233	234	235	236	237	238
239	240	241	242	243	244	245	246	247	248	249	250	251	252
253	254	255	256	257	258	259	260	261	262	263	264	265	266
267	268	269	270	271	272	273	274	275	276	277	278	279	280
281	282	283	284	285	286	287	288	289	290	291	292	293	294
295	296	297	298	299	300	301	302	303	304	305	306	307	308
309	310	311	312	313	314	315	316	317	318	319	320	321	322
323	324	325	326	327	328	329	330	331	332	333	334	335	336
337	338	339	340	341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360	361	362	363	364
365	366	367	368	369	370	371	372	373	374	375	376	377	378
379	380	381	382	383	384	385	386	387	388	389	390	391	392
393	394	395	396	397	398	399	400	401	402	403	404	405	406
407	408	409	410	411	412	413	414	415	416	417	418	419	420
421	422	423	424	425	426	427	428	429	430	431	432	433	434
435	436	437	438	439	440	441	442	443	444	445	446	447	448
449	450	451	452	453	454	455	456	457	458	459	460	461	462
463	464	465	466	467	468	469	470	471	472	473	474	475	476
477	478	479	480	481	482	483	484	485	486	487	488	489	490

Inquiry Numbers 491–980

491	492	493	494	495	496	497	498	499	500	501	502	503	504
505	506	507	508	509	510	511	512	513	514	515	516	517	518
519	520	521	522	523	524	525	526	527	528	529	530	531	532
533	534	535	536	537	538	539	540	541	542	543	544	545	546
547	548	549	550	551	552	553	554	555	556	557	558	559	560
561	562	563	564	565	566	567	568	569	570	571	572	573	574
575	576	577	578	579	580	581	582	583	584	585	586	587	588
589	590	591	592	593	594	595	596	597	598	599	600	601	602
603	604	605	606	607	608	609	610	611	612	613	614	615	616
617	618	619	620	621	622	623	624	625	626	627	628	629	630
631	632	633	634	635	636	637	638	639	640	641	642	643	644
645	646	647	648	649	650	651	652	653	654	655	656	657	658
659	660	661	662	663	664	665	666	667	668	669	670	671	672
673	674	675	676	677	678	679	680	681	682	683	684	685	686
687	688	689	690	691	692	693	694	695	696	697	698	699	700
701	702	703	704	705	706	707	708	709	710	711	712	713	714
715	716	717	718	719	720	721	722	723	724	725	726	727	728
729	730	731	732	733	734	735	736	737	738	739	740	741	742
743	744	745	746	747	748	749	750	751	752	753	754	755	756
757	758	759	760	761	762	763	764	765	766	767	768	769	770
771	772	773	774	775	776	777	778	779	780	781	782	783	784
785	786	787	788	789	790	791	792	793	794	795	796	797	798
799	800	801	802	803	804	805	806	807	808	809	810	811	812
813	814	815	816	817	818	819	820	821	822	823	824	825	826
827	828	829	830	831	832	833	834	835	836	837	838	839	840
841	842	843	844	845	846	847	848	849	850	851	852	853	854
855	856	857	858	859	860	861	862	863	864	865	866	867	868
869	870	871	872	873	874	875	876	877	878	879	880	881	882
883	884	885	886	887	888	889	890	891	892	893	894	895	896
897	898	899	900	901	902	903	904	905	906	907	908	909	910
911	912	913	914	915	916	917	918	919	920	921	922	923	924
925	926	927	928	929	930	931	932	933	934	935	936	937	938
939	940	941	942	943	944	945	946	947	948	949	950	951	952
953	954	955	956	957	958	959	960	961	962	963	964	965	966
967	968	969	970	971	972	973	974	975	976	977	978	979	980

Inquiry Numbers 981–1470

981	982	983	984	985	986	987	988	989	990	991	992	993	994
995	996	997	998	999	1000	1001	1002	1003	1004	1005	1006	1007	1008
1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022
1023	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036
1037	1038	1039	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050
1051	1052	1053	1054	1055	1056	1057	1058	1059	1060	1061	1062	1063	1064
1065	1066	1067	1068	1069	1070	1071	1072	1073	1074	1075	1076	1077	1078
1079	1080	1081	1082	1083	1084	1085	1086	1087	1088	1089	1090	1091	1092
1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103	1104	1105	1106
1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119	1120
1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134
1135	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148
1149	1150	1151	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162
1163	1164	1165	1166	1167	1168	1169	1170	1171	1172	1173	1174	1175	1176
1177	1178	1179	1180	1181	1182	1183	1184	1185	1186	1187	1188	1189	1190
1191	1192	1193	1194	1195	1196	1197	1198	1199	1200	1201	1202	1203	1204
1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215	1216	1217	1218
1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231	1232
1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246
1247	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260
1261	1262	1263	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274
1275	1276	1277	1278	1279	1280	1281	1282	1283	1284	1285	1286	1287	1288
1289	1290	1291	1292	1293	1294	1295	1296	1297	1298	1299	1300	1301	1302
1303	1304	1305	1306	1307	1308	1309	1310	1311	1312	1313	1314	1315	1316
1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327	1328	1329	1330
1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343	1344
1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358
1359	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372
1373	1374	1375	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386
1387	1388	1389	1390	1391	1392	1393	1394	1395	1396	1397	1398	1399	1400
1401	1402	1403	1404	1405	1406	1407	1408	1409	1410	1411	1412	1413	1414
1415	1416	1417	1418	1419	1420	1421	1422	1423	1424	1425	1426	1427	1428
1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439	1440	1441	1442
1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455	1456
1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470

(continued from page 63)

One popular type of audio compression, pioneered by the telecommunications industry, is LogPCM. This compression technique is based on a form of compression that is logarithmic instead of linear. One of two logarithmic functions are generally used: the μLaw function in Example 1(a), and the aLaw function in Example 1(b). The logarithmic nature of the compression algorithm provides nearly 2:1 compression by allowing the system to have 14 bits of dynamic range contained in a sample that only requires 8 bits to physically store. There are many devices currently available to implement this compression technique, and for the storage of voice-only signals, LogPCM is adequate to the task.

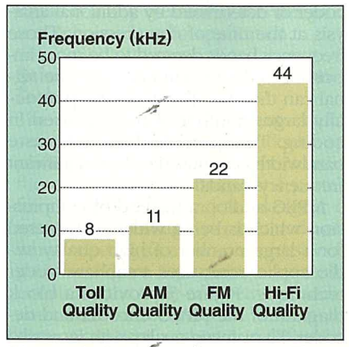


Figure 1: *Sampling frequency. The amount of data that will need to be stored for an audio signal is directly related to the quality of the output you wish to produce. Higher-quality outputs require more data to be stored and directly affect the type of compression technique that you should use.*

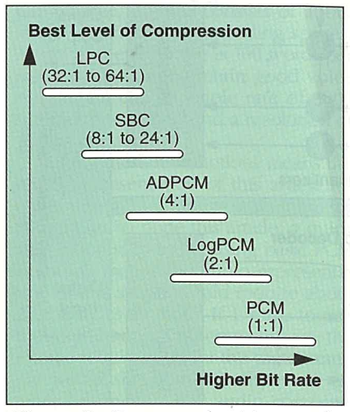


Figure 2: *Common algorithms and compression ratios.*

Another variation of PCM encoding, called "adaptive differential PCM" (ADPCM), gives a 4:1 compression ratio. It uses the history of the input audio signal as a method of predicting the changes in that signal. Instead of encoding the actual input audio signal's value, the ADPCM technique encodes the difference between the value of the input audio sample and the same signal's predicted value. If the prediction, which is typically done with a filter pro-

(a)

$$y(x) = \frac{V \log(1+\mu x/V)}{\log(1+\mu)}$$

(b)

$$y(x) = \begin{cases} \dfrac{Ax}{1+\log A}; 0 \le x \le \dfrac{V}{A} \\ \dfrac{V+V\log(Ax/V)}{1+\log A}; \dfrac{V}{A} \le x \le V \end{cases}$$

Example 1: *Logarithmic functions used in LogPCM compression: (a) the μLaw function; (b) the aLaw function.*

Consumer Information Catalog

Reading worth writing for.

If you're looking for some good reading, you've just found it. The free Consumer Information Catalog.

The Catalog lists about 200 federal publications, many of them free. They can help you eat right, manage your money, stay healthy, plan your child's education, learn about federal benefits and more.

So sharpen your pencil. Write for the free Consumer Information Catalog. And get reading worth writing for.

Consumer Information Center
Department RW
Pueblo, Colorado 81009

A public service of this publication and the Consumer Information Center of the U.S. General Services Administration.

THE ADVERTISER INDEX

RS#	Advertiser Name	Page #
1211	Academic Press	64
884	Analog Devices	47
*	*DDJ* CD ROM	33
872	Human Machine Interfaces	71
889	Intel Corporation	C2, 1
767	Iterated Systems	46
156	Metagraphics Software Corporation	35
1213	Metatech	64
874	Performance Computing	39
**	Product Debut	64
**	Programmer's Marketplace	79
501	Quarterdeck Office Systems	9
*	Raima Corporation	C3
1212	Siliconsoft	64
500	Software Interphase, Inc.	13
*	Software Publishing	C4
886	Time Arts	46
763	Van Nostrand Reinhold	23
*	Velocity Development	72

The index on this page is provided as a service to our readers. The publisher does not assume any liability for errors or omissions.

*The advertiser prefers to be contacted by phone; consult ad.
**See individual ads for RS#.

Advertising Sales Offices

Account Managers

Paul Miller	Northern California	(415) 358-9500
Lauren Sargeant	Southwest/Texas	(310) 472-4577
John Moon	Mid-Atlantic/Southeast	(508) 541-7425
Mary Kay Hoal	New England	(707) 557-1608
Stephanie Vandenberg	Northwest	(415) 358-9500
Eric Pitts	Midwest	(415) 358-9500
Cheryl Canion	Marketplace/Debut	(415) 358-9500

cess through a feedback loop, is good and the difference is small, the difference can easily be encoded with fewer bits. The quality of an audio input compression using the ADPCM technique is largely determined by the amount it is oversampled and the length of the signal's stored history.

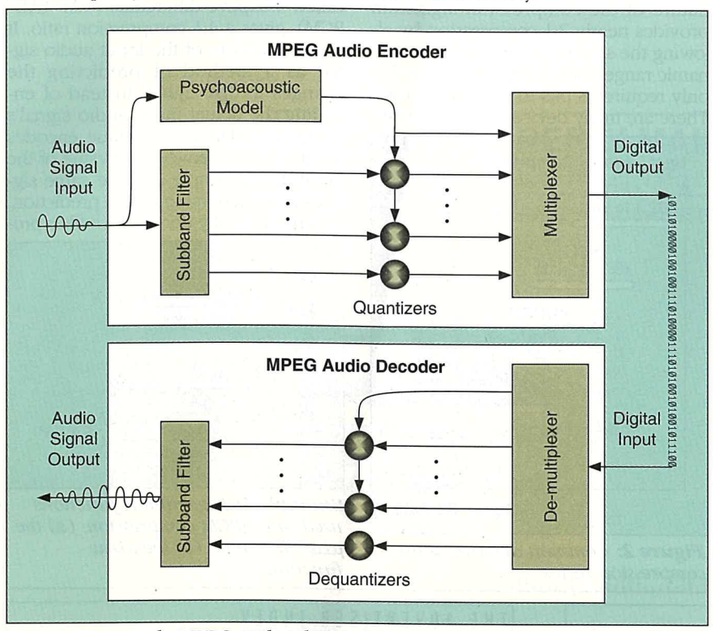


Figure 3: *A single MPEG audio channel. MPEG audio is a subband algorithm with adaptive quantization.*

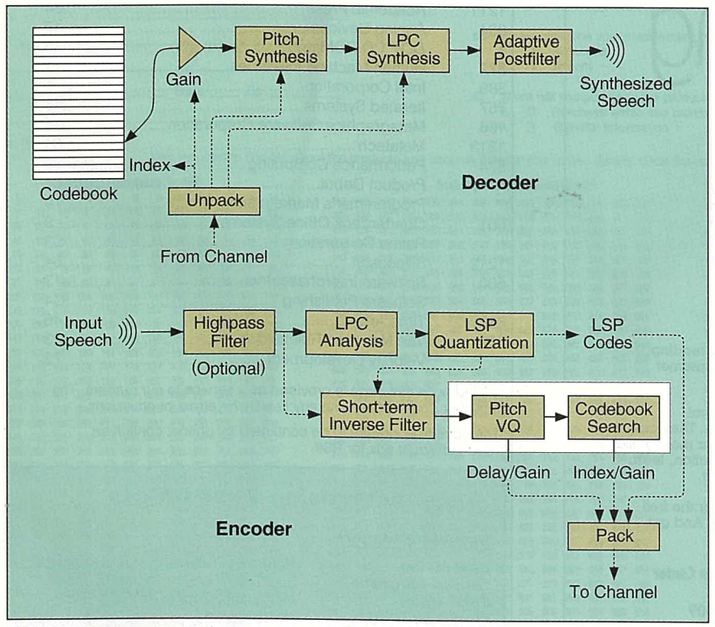


Figure 4: *CELP is a relatively new, enhanced version of LPC. This algorithm uses the specific qualities of human speech to provide maximum audio compression.*

The subband coder (SBC) can provide additional compression by combining elements of ADPCM with a technique for segmenting the signal into frequency bands. The significance of each frequency band is either predetermined for the coder or determined by additional analysis at the time of compression. Those frequency bands deemed to be more important to the overall quality of the signal can then be allocated a proportionally larger number of the bits used in coding. This process does not waste bandwidth on unused or less-significant frequency bands.

MPEG audio, a method of compression which is being widely considered for a large number of high-quality audio applications, uses a subband coder technique. Figure 3 provides a block diagram of the MPEG encoder and decoder. To maintain quality with increased compression, MPEG relies on a masking model. In this model, processing noise for one frequency band is covered up or masked by the signal in an adjacent frequency band. The use of multiple frequency bands and the masking model enables MPEG to be used for effective audio compression of signals with a 20-kHz audio bandwidth. Compression ratios as high as 24:1 can be realized with MPEG compression.

Linear predictive coding (LPC) is a technique which provides even greater levels of compression, but is applicable to voice rather than general audio. LPC was designed for human speech and uses a production model that takes advantage of speech-production characteristics—that is, the human vocal tract and the assumption of a single sound source. Using this model, compression ratios of 32:1 and greater are possible. One of the more recent variations of LPC, codebook-excited linear prediction (CELP), offers superior quality over previous versions. A block diagram of CELP is shown in Figure 4

Audio-compression Quality

While the quality of each compression technique shown in Figure 2 is subjective, there's general agreement that as the compression ratio increases, the quality decreases, at a given sample frequency. While this statement is generally true, there are certain exceptions based on the compression technique. In other words, linear PCM only maintains its superiority over LogPCM up to a point. Below that point (at fewer bits per sample), the noise introduced by reducing the number of bits in the PCM data is more noticeable than that introduced by the compression technique used by logPCM.

Basically, every compression technique has a limit on quality for different levels of compression. This means no single compression technique is a panacea for all potential applications. The selection of the proper audio-compression technique is a key element in the design of any enhanced audio application.

Selecting an Audio-compression Technique

The selection of the appropriate audio-compression model for a particular application is a complex process. None of the compression techniques is perfect for all applications, and the designer or system user must always make trade-offs when selecting an audio-compression algorithm. Figure 5 provides a means of visualizing the trade-offs between audio-compression techniques. The three-dimensional space shown is defined to be the complexity of the compression algorithm, the bit rate of the compressed data, and the audio, or acoustic, quality of the signal. In all three cases, the further the attribute is from the origin, the better it is. Although other attributes can be assigned to each of the axes, most are dependently linked to each other.

When a point on each axis is chosen, a plane in the three-dimensional space is defined. This plane remains relatively constant within the constraints of system cost, where system cost is defined in terms of processing power (MIPs) and memory requirements, both for processing and for storage or transmission. Therefore, to maintain a given level of audio quality, the system has to bear the additional cost of a higher bit rate or a more-complex algorithm. If the system cost cannot be increased, some trade-offs will have to be made between the algorithm's complexity and bit rate.

There are also several other dependent factors that you must consider when selecting a compression technique. The additional factors that affect the axes attributes are listed under their respective axes.

DSP-based Audio Compression

Although DSP techniques have been used in many fields for decades, it has only been since the early '80s, with the introduction of TI's TMS320 series of single-chip DSP devices, that these devices have had enough power to perform the calculations required to implement real-time audio-compression techniques for compression ratios greater than 2:1, while maintaining high levels of quality. Much of the early demand for DSP audio compression was driven by the telecommunications industry, where various techniques such as ADPCM have been in use for the last ten years.

A compelling example of just why DSP-based, or for that matter, any other type of audio compression, is important, can be made using this article as a test case. For the sake of argument, let's make the following assumptions: This article contains about 3000 words. On average, a word consists of about six ASCII characters, including spaces. A typical speaking rate is 100 words per minute. In order to retain good voice quality, I'll use a sample rate of 8000 samples per second and a resolution of 16 bits per sample.

Making these assumptions means the ASCII representation of this article requires nearly 20 Kbytes to store. If it takes 30 minutes to orate this article, a quick calculation shows that using basic PCM encoding means the audio representation of this article would require about 28.8 Mbytes to store! If I were to use the ADPCM compression technique, the storage requirements for this same example would shrink to 7.2 Mbytes. If the more-advanced CELP audio-compression technique were used, the audio representation of this article would become 900 Kbytes.

As Figure 6 demonstrates, while audio-compression techniques can vastly minimize the storage needed for a given sample, they can't reach the density of pure ASCII. The question you must ask is whether or not the advantages of audio outweigh the disadvantages, such as the associated overhead. Based on the advantages of incorporating audio into an application, the proper audio-compression technique makes this decision much easier to make.

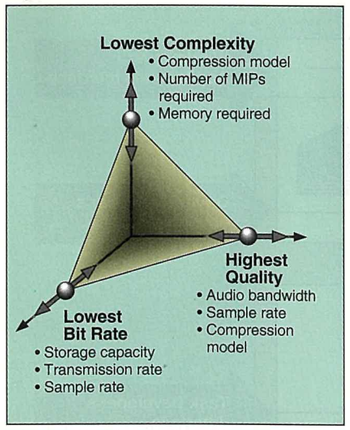


Figure 5: *The trade-offs between the basic elements of audio-compression algorithms modeled in three dimensions. If more than one of the dimensions is fixed at some upper bound, the attributes of the other dimensions must be varied and a compromise made.*

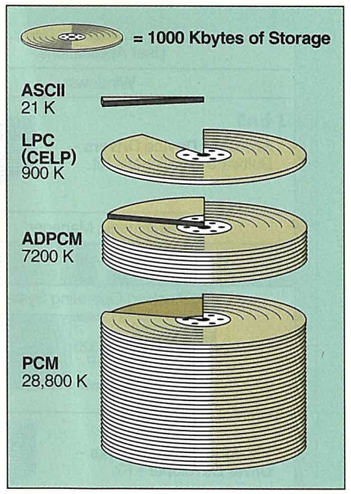


Figure 6: *Results of compressing a 3000-word article using various audio-compression techniques. PCM, ADPCM, and LPC assume 8K samples per second, 16 bits per sample.*

DSP and Audio Compression

An important aspect of the DSP-based system, especially as it relates to the various requirements of multimedia, is its ability to process and schedule multiple tasks simultaneously. The design of a flexible DSP-based system guarantees that each and every real-time task will have the processing resources it needs. As compression algorithms become more complex and the host processor becomes more overloaded, the need for a dedicated multitasking DSP-based solution becomes more pronounced.

Another important aspect of a DSP-based approach is the efficiency of its I/O system. By incorporating a powerful, direct memory access (DMA) capability, a DSP-based system is able to use most of its processing power for actual computation, not for moving data in and out of the system.

As previously mentioned, virtually all audio-compression techniques are typified by the MAC cycle. Since the architecture of the DSP has been defined to allow MAC cycles to occur efficiently while doing other tasks in parallel—such as fetching data from memory or storing results back to memory—audio-compression techniques map well onto DSP-based hardware. This "hand-in-glove" fit of a DSP-based architecture

to implement audio-compression algorithms is certainly more than coincidence. The latest techniques for audio compression, including MPEG and LPC, all require some sort of input filtering, such as infinite-impulse response (IIR), or finite-impulse response (FIR) filtering, in order to operate. This filtering process, which is very MAC intensive, is a perfect application for the DSP.

MPEG uses another key DSP technique, the Fast Fourier Transform (FFT), as an important part of its audio-compression algorithm. MPEG uses the FFT to determine where most of the energy in the input signal is and can therefore allocate the distribution of masking bits appropriately. Since the DSP supports real-time FFT processing, this distribution of masking bits can occur in real time, thus improving the overall compression algorithm and the output signal.

Software to Simplify System Development

The use of audio compression in a software application can seem like an overwhelming task, but in reality, a flexible, DSP-based solution simplifies this task significantly. Choosing the right system allows you to include audio-compression techniques without a corresponding increase in overall code complexity. This means you will be able to focus on the high-level metrics associated with audio compression; for example, the number of disk-storage bytes one minute of recorded speech or audio will require, or which sound quality is appropriate for a particular application. In order for this to take place, a simple interface to the DSP is a must.

A typical DSP-based system block diagram is shown in Figure 7. This block diagram, which is based on TI's TMS320 system, is typical of the standard DSP-based, multimedia solutions available today. A system such as this allows you to support many different types of audio compression without the overhead associated with the development of a new software driver for each one. Since the audio files are stored with information about the sample rate, the number of bits per sample, and the encoding algorithm, a DSP-based system allows a high-level API to be used. As new compression methods become available, the flexible nature of this type of system will accommodate them with a minimal impact on the system design.

As shown in Figure 7, there are two main software interfaces in a DSP-based system that are of interest to a system designer: the applications interface and the task developer's interface. The software support required for each of these, while very different, must be comprehensive to provide the system designer with the tools needed for a complete software-development environment.

Applications Interface

For most software developers, the applications interface — the interface between user applications, the operating system, and the DSP-based system — is most important. The key attribute for the applications interface is to provide a simple, standardized interface between the application and the DSP-based system. This allows you to easily incorporate advanced audio-compression techniques in new software applications and to make the use of different types of audio compression transparent to the application itself.

One method of accomplishing this is through the use of APIs to extend the base-level Windows Media Control Interface (MCI) device drivers. These high-level APIs would, for example, allow the application to simply call a driver that would play a compressed audio file. The driver itself recognizes the type of compression algorithm used and plays the file back accordingly. All of these operations would take place transparently to the user because of the advanced programmable nature of the DSP-based system. This means an application can easily incorporate audio compression to enhance its functionality and marketability.

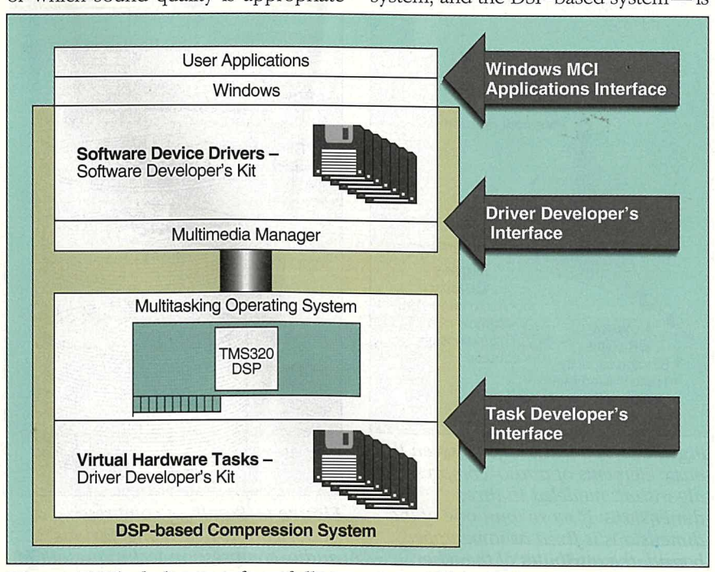


Figure 7: *Block diagram for a fully programmable, DSP-based audio-compression system based on Texas Instruments' TMS320 system.*

Task Developer's Interface

The task developer's interface is the low-level software interface to the DSP-based system that allows you to produce new applications-interface drivers and write new tasks (such as compression algorithms) for the DSP device itself. This level of software interface requires a full set of software-development tools, including debuggers, compilers, assemblers, and linkers as well as access to the DSP operating system.

The key to accessing this level of the system is the ability to incorporate the latest advancements in audio compression in your application without changing the hardware. By writing the underlying code associated with a new audio-compression algorithm, you can be assured that upgradability and functionality will follow your systems, increasing their value to the end user.

One example for isolating the functionality of a new process from the underlying hardware is shown in the code segment in Listing One, page 69. This task developer's model is relatively simple and provides a standard methodology for handling data I/O and process control. Data I/O handled in this manner minimizes the amount of overhead associated with the task of interfacing the system to data from a continuous source, or sink, such as A/D or D/A converters, by automatically keeping track of where the data is at all times.

Listing One contains two basic, key forms of task communications that are important to the task developer. The first involves the concept of interfacing a stream of continuous data, in either a compressed or uncompressed form, to the application via general-purpose connectors (GPCs), which are circular buffers that serve as the interface to the audio-compression algorithm (one GPC for the input, and one GPC for the output) and the I/O data streams. These circular buffers allow the algorithm to have free access to the data on an "as needed" basis without being burdened with the additional overhead and peculiarities associated with controlling the data transfers in and out of physical I/O channels.

The GPCs also provide an excellent means for inserting or removing additional tasks from a processing flow without requiring changes in each task to account for the presence or absence of the additional tasks. As an example, a µLaw compression task could be added following an LPC synthesis task if a µLaw CODEC is being used to convert the digital signals to analog in one system. In another system, where a linear CODEC is used, the µLaw compression task would be removed. In either case, the LPC synthesis task would process data in exactly the same way.

The second key concept involves the use of an intertask communications block (ITCB), which is a form of state control for system communications. The purpose of the ITCB is to pass pertinent information, such as task status and process commands, between the host processor and the DSP or between two DSP tasks. A typical application for an ITCB would be to store the volume-control information associated with the compressed-audio data. In this application, the ITCB would hold the audio-gain information for a particular data sample, which both the host processor and the DSP subsystem have direct, shared access to.

Listing One includes two functions, *Echo* and *ModifySample*, which demonstrate the processing that may be done on typical sample data. The *Echo* function simply demonstrates how to read data from the input GPC and write data to the output GPC. The purpose of *ModifySample* is to show where the data-compression algorithm would be placed in the code to implement the selected compression functionality.

Conclusion

The potential for audio compression in today's multimedia market is practically limitless. As more applications demand audio, practical system limitations, such as cost and system size, will dictate the need for audio compression. And while the current usage of enhanced audio in multimedia systems is impressive, it pales in comparison to the potential for future applications. Although the applications for enhanced audio compression continue to evolve, compression techniques are all based on fundamental concepts that will remain applicable well into the future. The selection of a programmable DSP-based system can help you implement the next generation of multimedia applications.

References

Lynch, Thomas J. *Data Compression: Techniques and Applications.* New York, N.Y.: Van Nostrand Reinhold, 1985.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 10.**

Listing One *(Text begins on page 63.)*

```
typedef struct gpc_t {
   void        *gpc_putp;       /* Get/Put Pointer */
   unsigned short    gpc_size;  /* Size of GPC, in bytes */
   unsigned short    gpc_mwpf;  /* Max. words to be written in 1 frame */
   void        **gpc_aput; /* Address of owner's put/user's get pointer */
   unsigned short    gpc_prot;  /* protocol to be used */
} gpc_t;

#define gpc_aget gpc_aput          /* Aliases for owner/user */
#define gpc_getp gpc_putp
   .
   .
   .
extern gpc_t vio_input, vio_output;
   .
   .
   .
extern struct vioitcb_t {
       int     hioctl;      /* @=>tel, -1=>handset connected to processor */
       int     inputctl;    /* @=>handset, -1=>microphone input */
       int     shactive;    /* @=>on-hook, -1=>off-hook */
} *vioitcb2;

 /*
  * prototypes for routines declared later
  */
void Echo(gpc_t * src, gpc_t * dst, short n);
int ModifySample (short s);
   .
   .
   .
short oh;
void main(void)
{
        vioitcb2->hioctl   = actl;
        vioitcb2->inputctl = amic;
        oh =  vioitcb2->shactive;
        if (oh) {                       /* Are we off-hook? */
                Echo(&vio_output, &vio_input, VIO_SPF);
        } else {
                GPCFillM128(&vio_input, VIO_SPF, 0);
                GPCAdvance32M128(&vio_output);
        }
}
  /*
   * Echo
   *
   *   src is a pointer to the input GPC
   *   dst is a pointer to the output GPC
   *   n is the number of words (16-bit) to move
   */

 void Echo(gpc_t * src, gpc_t * dst, short n)
 {
        short *s;
        short *d;
        short sample;
        s = src->gpc_getp;
        d = dst->gpc_putp;
        while (n-- > 0) {
                s = GPCIncM128(s);
                sample = *s;
                /* NOW MODIFY THE SAMPLE */
                sample = ModifySample (sample);
                d = GPCIncM128(d);
                *d = sample;
        }
        src->gpc_getp = s;
        dst->gpc_putp = d;
 }
   .
   .
   .
```

End Listing

Build Your Own RS-232 Sound System

Digital audio from your RS-232 port

Dennis Cronin

Have you ever thought about using the extra RS-232 port on your computer as an audio output? Probably not, right? However, it can be done. This article describes a rather unique method of coding data that will produce an audio signal when streamed to a speaker attached to the RS-232 port. Granted, the sound isn't CD quality, but it's still quite intelligible. I'll start by examining how to get an audio signal from what was only designed to be a serial data port. Then I'll look at a program that outputs a Microsoft Windows .WAV audio file to the comm port of your PC. Finally, I'll provide the wiring diagrams necessary to build your own speaker to attach to the comm port of your PC.

Dennis almost completed an EE degree before being lured into the sordid world of fast computers, easy money, and loose connections. He currently specializes in UNIX driver development for Solaris and HP-UX operating systems and can be contacted at denny@cd.com.

From ASCII to Audio

To understand how to turn a mundane ASCII character stream into music, you need to look beneath the characters, at the underlying serial bit stream coming from the UART. Although a complete primer on asynchronous communications is beyond the scope of this article, I'll cover some basics of where the audio comes from.

An RS-232 asynchronous communications line idles at a state defined as being the "1" state of the line. This is also referred to as a "marking" state. When you shove, say, the letter *A* into the transmit register of the UART, it generates a start bit. A start bit is always a 0, and the transition from the marking state of 1 to that first start bit of 0 is the critical point of synchronization whence all the rest of the bit stream is referenced.

Figure 4: *New highs in low-fi: a roll-your-own speaker.*

After the start bit, the UART proceeds to send the bits of the character, starting with the least-significant bit (LSB). An *A* is 0x41, so the LSB is a 1. Assuming that the line settings are for 8 bits and no parity, you'll see the UART crank out all 8 bits of the character, immediately followed by a stop bit, which is always a 1. The stop bit is also a critical point, since the receiving UART uses it as a simple check to make sure synchronization was maintained throughout the reception of the character. If it doesn't see a 1 in the stop-bit slot, it reports the familiar "framing error" status. Figure 1 shows what the letter *A* looks like as it emerges from the UART, all framed up with start and stop bits.

While the letter *A* is interesting enough, the character *U* is *really* interesting. As you can see in Figure 2(a), *U* (defined as 0x55) happens to generate a completely alternating bit pattern, including the start and stop bits. When a steady stream of characters is transmitted, the stop bits are immediately followed by the start bit of the next character. Figure 2(b) shows that a steady stream of *U* characters is actually just a continuous square wave, the frequency of which is simply half the baud rate programmed at the UART.

If you were to hook up your speaker to a UART while it generated a steady stream of *U*s, you'd hear a high-pitched tone as this square wave wiggled the speaker back and forth. But suppose you set the baud rate up very high, say at 115,200 baud, which is the fastest baud rate commonly available on PCs. Well, the speaker can't wiggle that fast, nor can we humans hear a frequency that high. Instead, the speaker just sees the effective average value of the waveform, in this case, 0 volts, since there is an equal distribution of 1s and 0s

Now, suppose you pop a character with an extra bit turned on into the stream of *U*s every once in a while. Your trusty *U*s contain four 1s and four 0s, so pick a character that has five 1s and three 0s — a *W* (0x57), for instance. When you shove the *W* in, the speaker briefly sees an imbalance in the quick, alternating tugs the stream of *U*s provides. The response to this brief imbalance manifests itself as an exciting "click" every time you shift out a *W*. But this isn't quite music yet.

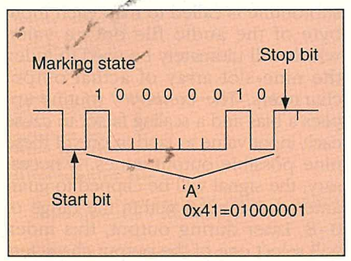


Figure 1: *The letter A, as seen emerging from the UART, all framed up with start and stop bits.*

What happens when you turn on six 1s, and only do two 0s? The speaker will be a little more imbalanced and will move a little farther. How about seven 1s and one 0? It turns out that you can "unbalance" that speaker by four distinct values on each side of the center value created by the steady stream of *U*s. Thus, you have nine "positions" you can ask the speaker to assume depending on the mix of 1s and 0s you send it. It's a little DAC (digital-to-analog converter), albeit a humble, 3-bit one (plus the zero position). By picking the right character, you can control the movement of the speaker.

So how many "samples" per second can your DAC do? With 8 bits per character, plus the start and the stop bit, you have a total of 10 bits for each character. At the 115,200-baud rate, this works out to 11,520 characters (or samples) per second. So your DAC is capable of sustaining a sample rate of just over 11K samples per second, which isn't quite CD quality, but is still good enough for voice-grade audio.

In most digital-audio systems, the DAC is immediately followed by a sharp-cutoff lowpass filter. This prevents aliasing and provides the necessary smoothing function to turn the discrete steps back into smooth analog. You're going to have to forgo this luxury and settle for the simple lowpass action resulting from the speaker's mechanical inertia. It doesn't matter that much. The three bits, plus 0 output resolution is still the biggest limiting factor in terms of over-

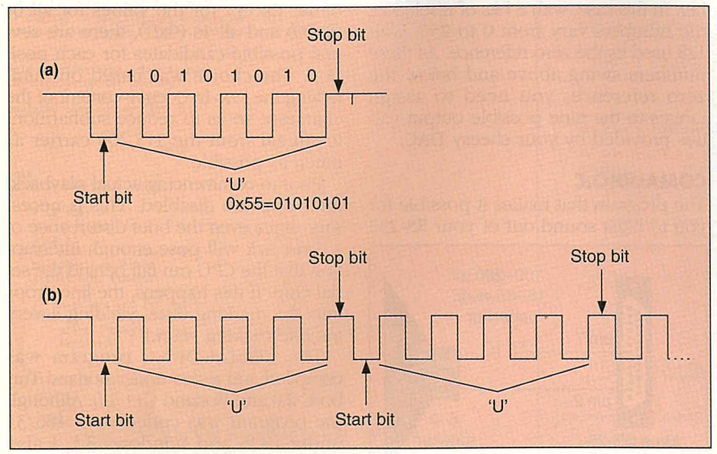
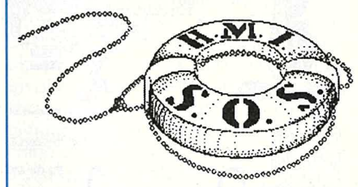


Figure 2: *(a) The letter* U *(defined as 0x55) generates a completely alternating bit pattern, including the start and stop bits; (b) a steady stream of* U *characters is a continuous square wave, the frequency of which is half the baud rate programmed at the UART.*

Sound Operating System

The complete Digital Audio and MIDI solution.



➢ Compatible

Sound Blaster	Sound Blaster Pro
Thunderboard	Pro Audio Spectrum/16
Adlib Gold 1000	Microsoft Sound System
Tandy Sensation	Gravis UltraSound
Sound Master II	Disney Sound Source
Adlib Digitized	PC Internal Speaker
UART MIDI	MPU-401
OPL2	OPL3

➢ Digital Audio Features

- ✓ 8 Channel **real-time** mixing
- ✓ Auto-detection of devices
- ✓ 8 & 16 bit data support
- ✓ Independent volume control
- ✓ Stereo/Mono data support
- ✓ Recording & Playback

➢ MIDI Features

- ✓ MIDI 1.0 compatible
- ✓ Digitized instruments, ie. **DRUMS**
- ✓ Dynamic Voice Allocation
- ✓ Simultaneous **Multi-Song** Play
- ✓ System exclusive dump
- ✓ **Video/Graphics** synchronization

➢ Complete System

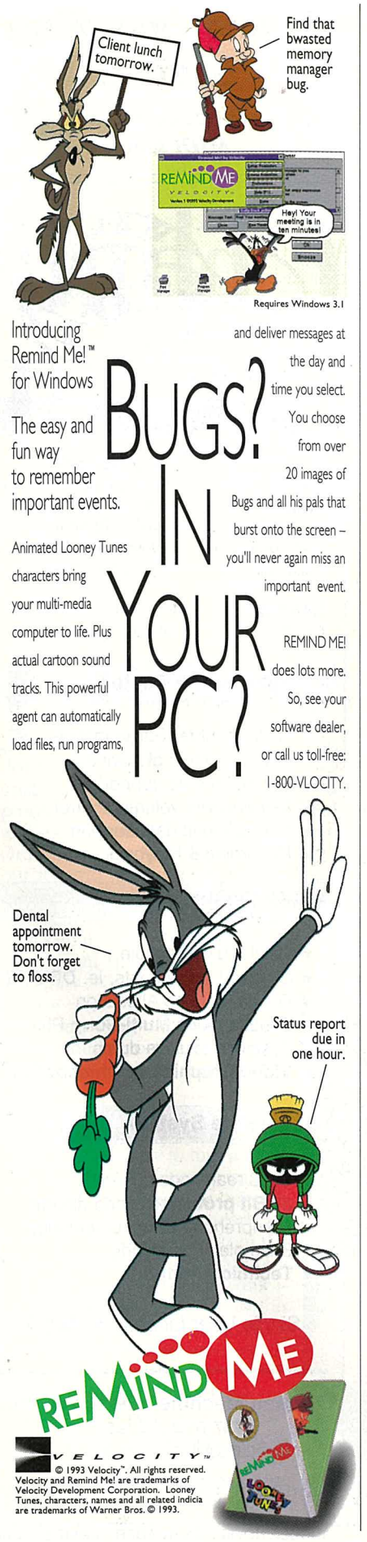
- ✓ DOS real mode drivers
- ✓ **32 Bit protected** mode drivers
- ✓ Comprehensive documentation
- ✓ Example source code
- ✓ **Technical support**

Single User License **$199.00**

Human Machine Interfaces, Inc.
757 West N Street
Springfield Oregon 97477

Tel: (503) 747-2314 Fax: (503) 747-2627

CIRCLE NO. 872 ON READER SERVICE CARD


Client lunch tomorrow.
Find that bwasted memory manager bug.
REMIND ME
VELOCITY
Hey! Your meeting is in ten minutes!
Requires Windows 3.1
Introducing Remind Me!™ for Windows
The easy and fun way to remember important events.
BUGS? IN YOUR PC?
Animated Looney Tunes characters bring your multi-media computer to life. Plus actual cartoon sound tracks. This powerful agent can automatically load files, run programs, and deliver messages at the day and time you select. You choose from over 20 images of Bugs and all his pals that burst onto the screen – you'll never again miss an important event.
REMIND ME! does lots more. So, see your software dealer, or call us toll-free: 1-800-VLOCITY.
Dental appointment tomorrow. Don't forget to floss.
Status report due in one hour.
REMIND ME
VELOCITY™
© 1993 Velocity™. All rights reserved. Velocity and Remind Me! are trademarks of Velocity Development Corporation. Looney Tunes, characters, names and all related indicia are trademarks of Warner Bros. © 1993.

all "fidelity" (and I use that term very loosely).

Now that you have an understanding of how to coax some audio out of the RS-232 port, let's look at what you have to do to actually play the Microsoft Windows WAV files.

Playing WAV Files Over RS-232

The most common WAV file format contains 8-bit mono audio data in linear pulse code modulation (PCM) format using a sample rate of 11,000 samples per second. Other formats are possible, but so far this is the most common. The 11-kHz sample rate happens to be close enough to our own 11,520 DAC sample rate, that a sample-rate conversion isn't even necessary. Sounds will play back slightly less than a semitone too high, but that's close enough for rock 'n' roll.

Information about the actual file format is contained in a RIFF header at the front of the file. For simplicity's sake, you're just going to ignore that header and assume the previously stated sound-file parameters. Well, you're not going to completely ignore the header…in fact, you're going to "play" it. The header is so short (in terms of audio data) that the click it adds to the beginning of the sound is almost imperceptible. Since the program is, by definition, not particularly hi-fi, it's just not worth the extra effort to parse that header, so we're going to make a KISS design decision right off that bat. (Header? I don't see no header.)

Linear PCM is probably the simplest audio format to deal with, as each instantaneous value of the audio waveform is directly represented by a number. In this case, with 8 bits of resolution, the numbers vary from 0 to 255, with 128 used as the zero reference. As these numbers swing above and below the zero reference, you need to assign ranges to the nine possible output values provided by your cheesy DAC.

COMAUDIO.C

The program that makes it possible for you to blast sound out of your RS-232 port is COMAUDIO.C (Listing One, page 73). The area of COMAUDIO.C that's of interest is in the conversion of the 8-bit PCM value to the character for output to the serial chip. As you read the audio file in, the *convert()* subroutine is called to map each input byte of the audio file onto a value which will ultimately be used to index the nine-slot array of actual output characters. The *convert()* routine applies a bias and a scaling factor to cause each input value to land in one of these nine possible output zones. If necessary, the signal will be clipped to guarantee that it stays within the range of 0–8. Later during output, this index will select one of the output characters from the *dac[9]* array.

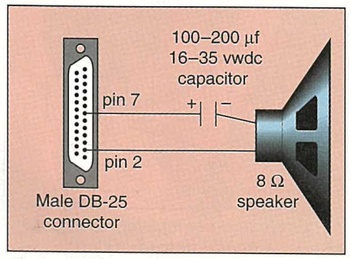


Figure 3: *Wiring diagram for the speaker.*

The most common WAV file contains 8-bit mono audio data in PCM format, using a sample rate of 11 kHz

The characters in the *dac[9]* array are arranged in order of increasing numbers of 1s. Except for the values for all 0s (0x00) and all 1s (0xff), there are several possible candidates for each position. The choice was based on minimizing the low-frequency content of the character, so as to reduce subharmonic squeal from the 115,200 carrier as much as possible.

Prior to commencing actual playback, interrupts are disabled. This is necessary, since even the brief disturbance of a timer tick will pose enough interruption that the CPU can fall behind the serial chip. If this happens, the line drops into the marking state, yielding a very audible clicking sound.

The COMAUDIO.C program was compiled and tested under Borland Turbo C 2.0 and Borland C++ 1.0. Although the program was coded on a 486/33 under DOS and Windows 3.1, I also tested it successfully on 286-class PCs. Similar versions of COMAUDIO.C have been tested on a SPARC 1+ under SunOS 4.1.3. If you have a slower machine, you might need to do some optimizing of the output function. (Or better yet, you might just buy a new computer. Jeez, get into the '90s.) After you compile the program, the only thing left to do is hook up a speaker. Sample data files in .WAV, .C3P, and .AU (Sun) format are available electronically; see "Availability," page 2.

Attaching the Speaker

Figure 3 is a wiring diagram for the speaker. While just about any speaker will work, a cheap replacement 8-ohm speaker (Radio Shack 40-1208 or equivalent) is ideal. You actually want the frequency response to be somewhat limited in order to help filter out some of the nasty, high-frequency digital hash.

A capacitor of at least 100 µf, with a voltage rating of at least 16 WVDC is put in series with the speaker to block DC. Polarity is important; make sure you get the negative terminal of the capacitor attached to the speaker and the positive terminal attached to pin 7 of the DB-25 connector.

The speaker will sound somewhat better if you enclose it in something. A simple cardboard box works well enough; see Figure 4. Plus, this keeps your friends guessing about what complex circuitry, that's capable of magically turning the RS-232 into audio, lurks inside.

Hook the speaker up to one of the comm ports on your PC and give it a whirl. Unless you've been living on a deserted isle, you can probably locate a copy of TADA.WAV, a small sound file that ships with Windows 3.1, to use as a test. Run COMAUDIO with TADA.WAV as an argument, specifying the proper comm port if necessary, and you should hear delightful, although brief, strains of music emanating from your proud little speaker.

ARPEGGIO.C

In addition to the WAV files that are available electronically, there are plenty of places you can download more. But if you want something a little more melodious to play right away, ARPEGGIO.C (Listing Two, page 74) can be used to generate a sample audio file for playback through COMAUDIO. Simply invoke it with a destination filename, and it will write out an audio file containing a catchy little ditty. If your machine doesn't have a math coprocessor, it can take a couple of minutes or more to generate the output file, so be patient. ARPEGGIO.C is easy to modify, and you can change it to generate electronic compositions of your own.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 11.**

Listing One *(Text begins on page 70.)*

```
/*
    COMAUDIO.C - uses PC com port to generate audio
*/
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <sys/stat.h>
#include <dos.h>

/* defs for low level access to serial chip */
#define SCC_DATA    0
#define SCC_INTCTRL 1
#define SCC_CTRL    3
#define SCC_STATUS  5
#define TXRDY       0x20
#define comout(scc_base,c)                              \
{                                                       \
    while((inportb(scc_base + SCC_STATUS) & TXRDY) == 0);   \
    outportb(scc_base + SCC_DATA,c);                    \
}

/* farinc - macro to increment far ptr */
#define farinc(p) {                                     \
                    p++;                                \
                    if(FP_OFF(p) == 0)                  \
                        p = MK_FP(FP_SEG(p) + 0x1000,0);    \
                  }

/* digital to ASCII analog conversion table */
unsigned char dac[9] = {0x00,0x08,0x12,0x29,0x55,0x6b,0xb7,0xef,0xff};

/* protos */
void main(int argc, char **argv);
int line_setup(int linenum);
void set_vol(int volume);
unsigned char convert(int c);

/*
                        main
*/
void
main(int argc,char **argv)
{
    FILE *fp;
    unsigned char far *p, far *bufp;
    long i;
    int port = 1, volume = 5;
    register c,scc_base;
    struct stat statbuf;

    /* check arg cnt for sanity */
    if(argc < 2 || argc > 4) {
        printf("Usage: comaudio [wavfile] [[port]] [[volume]]\n");
        exit(1);
    }

    /* if com port spec'd */
    if(argc > 2) {
        port = atoi(argv[2]);
        if(port < 0 || port > 1) {
            printf("Use 0 or 1 to select com1 or com2 respectively.\n");
            exit(1);
        }
    }

    /* see if volume is spec'd */
    if(argc > 3) {
        volume = atoi(argv[3]);
        if(volume < 1 || volume > 9) {
            printf("Volume should be in range 1-9.\n");
            exit(1);
        }
    }
    set_vol(volume);

    /* get length of sound file */
    if(stat(argv[1],&statbuf) != 0) {
        printf("Cannot stat sound file '%s'\n",argv[1]);
        exit(1);
    }

    /* try to alloc mem to hold entire (nibble packed) file */
    bufp = farmalloc(statbuf.st_size / 2);
    if(bufp == NULL) {
        printf("Cannot allocate %lu bytes of memory for sound file\n",
            statbuf.st_size / 2);
        exit(1);
    }

    /* open sound file */
    fp = fopen(argv[1],"rb");
    if(fp == NULL) {
        printf("Cannot open sound file '%s'\n",argv[1]);
        exit(1);
    }

    /* read entire file into mem */
    for(i = statbuf.st_size / 2, p = bufp; i--; ) {
        /* pack 2 converted vals per byte */
        c = convert(fgetc(fp));
        *p = c | (convert(fgetc(fp)) << 4);
        farinc(p);
    }

    /* set up port */
    scc_base = line_setup(port);

    /* grab from buf and shove out com port */
    disable();      /* ints must be off for full "fidelity" */
    for(i = statbuf.st_size / 2, p = bufp; i--; ) {
        /* unpack to vals per byte */
        c = *p;
        comout(scc_base,dac[c & 0xf]);
        comout(scc_base,dac[c >> 4]);
        farinc(p);
    }
    enable();       /* turn interrupts back on */
    exit(0);
}

/*
                        line_setup

    Sets up spec'd line for 115200, returns ptr to
    assoc'd chip channel.
*/
int
line_setup(int linenum)
{
    union REGS regs;
    int scc_base;

    scc_base = linenum ? 0x2f8 : 0x3f8;
```

(continued on page 74)

Listing One *(Listing continued, text begins on page 70.)*

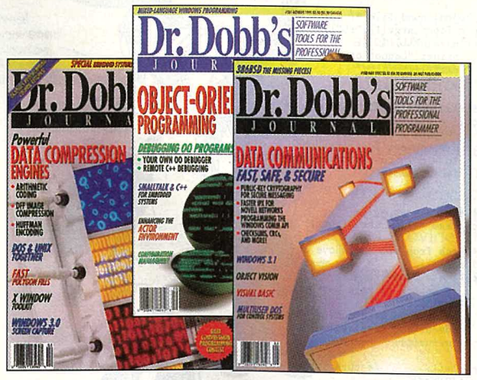
```
    /* BIOS call does most of it */
    regs.h.ah = 0;
    regs.h.al = 0xe3;                   /* 9600,N,8,1 */
    regs.x.dx = linenum;
    int86(0x14,&regs,&regs);

    /* now talk nasty to the chip, write baud div = 1 for 115200 */
    outportb(scc_base + SCC_CTRL,inportb(scc_base + SCC_CTRL) | 0x80);
    outportb(scc_base + SCC_DATA,1);        /* write least sig */
    outportb(scc_base + SCC_INTCTRL,0);     /* write most sig */
    outportb(scc_base + SCC_CTRL,inportb(scc_base + SCC_CTRL) & 0x7f);
    return(scc_base);
}

static atten,bias;

/*
                        set_vol

    Sets conversion factors for spec'd volume.
*/
void
set_vol(int volume)
{
    atten = (10 - volume) * 3;
    bias = (256 - (9 * atten)) / 2;
}

/*
                        convert

    Converts 8 bit PCM value to index into ASCII lookup table
    by attenuating and clipping as necessary.

*/
unsigned char
convert(int c)
{
    c -= bias;
    if(c < 0) c = 0;    /* clip negative peaks */
    c /= atten;
    if(c > 8) c = 8;    /* clip positive peaks */
    return(c);
}
```

Listing Two

```
/*
        ARPEGGIO.C - generates test sound file
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define SAMPRATE    11520           /* for 115.2 kbaud */
#define DURATION    (SAMPRATE/15)   /* 1/15 sec units */
#define MAX_NOTES   73              /* six octaves */

FILE *fp;       /* output file */

/* protos */
void init_notes(void);
void note(int note_num, int duration, int decay);
void arpeggio(int basenote, int step);

/*
                        main
*/
main(int argc, char **argv)
{
    if(argc != 2) {
        printf("Usage: arpeggio [outfile]\n");
        exit(1);
    }
    if((fp = fopen(argv[1],"wb")) == NULL) {
        printf("Can't open output file '%s'\n",argv[1]);
        exit(1);
    }
    init_notes();

    arpeggio(24,0);
    arpeggio(15,12);
    arpeggio(22,12);
    arpeggio(19,0);
    arpeggio(24,0);
    note(24,10,8);
    exit(0);
}

/*
                        arpeggio

    Recursively generates a pretty little arpeggio.
*/
void
arpeggio(int basenote, int step)
{
    note(basenote + step,1,2);  /* plinky going up */
    if(step == 24)              /* 2 octave arpeggio */
        return;
    else if(step % 12 == 4 || step % 12 == 9)
        arpeggio(basenote,step + 3);
    else
        arpeggio(basenote,step + 2);
    note(basenote + step,1,8);  /* legato coming down */
}

static double notetab[60], rad_per_samp;

/*
                        init_notes

    Builds note frequency table and calcs radians/sample.
*/
void
init_notes(void)
{
    double twlfth_root2,freq;
    int i;

    twlfth_root2 = pow(2.0,1.0 / 12.0); /* compute semitone interval */
    for(i = 0, freq = 110.0 ; i < MAX_NOTES; i++) {
        notetab[i] = freq;
        freq *= twlfth_root2;           /* up a semitone */
    }
    rad_per_samp = 2.0 * M_PI / SAMPRATE;
}

/*
                        note

    Looks up note in frequency table and performs.
*/
void
note(int note_num, int duration, int decay)
{
    int c;
    long i,cnt;
    double freq,val,env,vol = 50.0;

    freq = notetab[note_num];               /* look up note frequency */
    cnt = duration * DURATION;              /* calc count for duration */
    env = 0.999 + decay * 0.0001;           /* calc envelope decay factor */
    for(i = 0; i < cnt; i++) {
        val = sin(rad_per_samp * freq * i); /* compute sine wave val */
        c = (int)(vol * val) + 128;         /* convert to 8 bit PCM */
        fputc(c,fp);                        /* write to output file */
        vol *= env;                         /* make note decay */
    }
}
```

End Listings

DID YOU MISS THESE?



BACK ISSUES

Call now to order your back issues of *DR. DOBB'S JOURNAL*:

(800) 444-4881

Mail Orders: P.O. Box 105448, Atlanta, GA 30348-5448.
Copies are $7 each, Canada & Mexico $9.00; all others $11.
Sales tax required on all orders for NY, MA, GA, and CA.
For foreign orders call 1-404-426-1926.
Call for volume discount prices.

HURRY SUPPLIES LIMITED

Listing One *(Text begins on page 4.)*

```
/*************************************************************
* FILE: linecalc.c
* DESC: Warping calculations and line-handling functions.
* HISTORY:  Created   3/11/1993   LAST CHANGED:  5/ 6/1993
*   Copyright (c) 1993 by Scott Anderson
*************************************************************/

/* ----------------------INCLUDES---------------------------- */
#include <conio.h>
#include <stdio.h>
#include <io.h>
#include <math.h>
#include <graph.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include "define.h"

/* ----------------------MACROS------------------------------ */
#define PIXEL(p,x,y)    (p->pixmap[y * (long) p->wide + x])
#define SQUARE(x)       (((long) x)*(x))

/* ----------------------PROTOTYPES-------------------------- */
/**** line routines ****/
int     xorLine(int x1, int y1, int x2, int y2);
int     getLine(int *argx1, int *argy1, int *argx2, int *argy2);
int     findPoint(LINE_LIST *lineList, int * line, int * point, int x, int y);
int     movePoint();
/**** warping and morphing routines ****/
int     sumLines(PICTURE *picture, COLOR *color, LINE *origline,
                                                    POINT *warp, LINE *warpline);
float   getSourceLoc(POINT *orig, LINE *origline, POINT *warp, LINE *warpline);
int     setLength(LINE *line);
void    setupScreen(PICTURE *pic, int editFlag);

/* ----------------------EXTERNALS--------------------------- */
/* set from last picture loaded */
extern int      Xmin, Ymin, Xmax, Ymax;
extern int      NumLines;
extern LINE     SrcLine[MAX_LINES];
extern LINE     DstLine[MAX_LINES];

/* ----------------------GLOBAL DATA------------------------- */
int     TargFlag=0;
/********  These are the basic warping calculations   **********/
* FUNC: int sumLines(PICTURE *picture, COLOR *color,
*                    LINE *origline, POINT *warp, LINE *warpline)
* DESC: Sum and weight the contribution of each warping line
*************************************************************/
int
sumLines(PICTURE *picture, COLOR *color, LINE *origline,
                                                    POINT *warp, LINE *warpline)
{
    int     x, y;
    float   weight, weightSum;
    float   distance;
    int     line;
    POINT   orig;
    int     paletteIndex;
    float   deltaSumX = 0.0;
    float   deltaSumY = 0.0;
    /* if no control lines, get an unwarped pixel */
    if (NumLines == 0)
        orig = *warp;
    else {
        weightSum = 0.0;
        for (line = 0; line < NumLines; line++, origline++, warpline++) {
            distance = getSourceLoc(&orig,origline,warp,warpline);
            weight = 1/(.001+distance*distance);
            deltaSumX += (orig.x - warp->x) * weight;
            deltaSumY += (orig.y - warp->y) * weight;
            weightSum += weight;
        }
        orig.x = warp->x + deltaSumX / weightSum + .5;
        orig.y = warp->y + deltaSumY / weightSum + .5;
    }
    /* clip it to the nearest border pixel */
    x = clip(orig.x, Xmin, Xmax);
    y = clip(orig.y, Ymin, Ymax);
    paletteIndex = PIXEL (picture, x, y);
    color->r = picture->pal.c[paletteIndex].r;
    color->g = picture->pal.c[paletteIndex].g;
    color->b = picture->pal.c[paletteIndex].b;
    return (paletteIndex);
}
/*************************************************************
* FUNC: float getSourceLoc(POINT *orig, LINE *origline,
*                          POINT *warp, LINE *warpline)
* DESC: For a given line, locate the corresponding warped pixel
*************************************************************/
float
getSourceLoc(POINT *orig, LINE *origline, POINT *warp, LINE *warpline)
{
    float fraction, fdist;
    int dx, dy;
    float distance;
    dx = warp->x - warpline->p[0].x;
    dy = warp->y - warpline->p[0].y;
    fraction = (dx * (long) warpline->delta_x + dy
            * (long) warpline->delta_y) / (float) (warpline->length_square);
    fdist = (dx * (long) -warpline->delta_y + dy
            * (long) warpline->delta_x) / (float) warpline->length;
    if (fraction <= 0 )
        distance = sqrt(dx*(long) dx + dy * (long) dy);
    else if (fraction >= 1) {
        dx = warp->x - warpline->p[1].x;
        dy = warp->y - warpline->p[1].y;
        distance = sqrt(dx*(long) dx + dy * (long) dy);
    }
    else if (fdist >= 0)
        distance =  fdist;
    else
        distance = -fdist;
    orig->x = origline->p[0].x + fraction * origline->delta_x -
            fdist * origline->delta_y / (float) origline->length + .5;
    orig->y = origline->p[0].y + fraction * origline->delta_y +
            fdist * origline->delta_x / (float) origline->length + .5;
    return distance;
}
/*************************************************************
* FUNC: int setLength(LINE *line)
* DESC: Set the deltas, the length and the length squared for a given line.
*************************************************************/
int
setLength (LINE *line)
{
    line->delta_x = line->p[1].x - line->p[0].x;
    line->delta_y = line->p[1].y - line->p[0].y;
    line->length_square = SQUARE(line->delta_x) + SQUARE(line->delta_y);
    line->length = sqrt(line->length_square);
}
/*********************  The line routines  *********************/
* FUNC: int xorLine(int x1, int y1, int x2, int y2)
* DESC: Draw a line on the screen using the XOR of the screen index.
*************************************************************/
int
xorLine(int x1, int y1, int x2, int y2)
{
    int oldcolor = _getcolor();
    _setcolor(WHITE);   /* Use white as the xor color */
    _setwritemode(_GXOR);
    _moveto (x1,y1);
    _lineto (x2,y2);
    _setcolor(oldcolor);    /* restore the old color */
}
/*************************************************************
* FUNC: int getLine(int *argx1, int *argy1, int *argx2, int*argy2)
* DESC: Input a line on the screen with the mouse.
*************************************************************/
int
getLine (int *argx1, int *argy1, int *argx2, int *argy2)
{
    int     x1,y1, x2,y2;
    int     oldx, oldy;
    int     input;
    /* save the current mode */
    short   old_mode = _getwritemode();
    /* get input until we have a real line, not just a point */
    do {
        /* wait for button or key press */
        while (!(input = mousePos (&x1, &y1)));
        if (input & KEYPRESS) {
            _setwritemode(old_mode);
            return 1;
        }
        oldx=x1, oldy=y1;
        hideMouse();
        /* prime the pump with this dot */
        xorLine (x1, y1, oldx, oldy);
        showMouse();
        while (input = mousePos (&x2, &y2)) {
            /* rubber band a line while the mouse is dragged */
            if (x2 != oldx || y2 != oldy)
            {
                hideMouse();
                xorLine (x1, y1, oldx, oldy);
                xorLine (x1, y1, x2, y2);
                showMouse();
                oldx=x2, oldy=y2;
            }
        }
    } while (x1 == x2 && y1 == y2);
    *argx1 = x1, *argy1 = y1;
    *argx2 = x2, *argy2 = y2;
    _setwritemode(old_mode);        /* get out of XOR mode */
    return (0);
}
/*************************************************************
* FUNC: int findPoint(LINE_LIST *lineList,int * line,int * point,int x, int y)
* DESC: loop thru dstline and find point within GRAB_DISTANCE,
*       return 1 if found, 0 otherwise.
*************************************************************/
int
findPoint (LINE_LIST *lineList, int * line, int * point, int x, int y)
{
    int l, p;
    int minl, minp;
    long length;
    long minlength = SQUARE(640) + SQUARE(480);
    for (l = 0; l < lineList->number; l++) {
        for (p = 0; p <= 1; p++) {
            length = SQUARE(lineList->line[l].p[p].x - x)
                   + SQUARE(lineList->line[l].p[p].y - y);
            if (length < minlength) {
                minlength = length;
                minl = l;
                minp = p;
            }
        }
    }
    if (minlength > GRAB_DISTANCE)
        return 0;
    *line = minl;
    *point = minp;
    return 1;
}
/*************************************************************
* FUNC: int movePoint(LINE_LIST *lineList)
```

(continued on page 76)

Listing One *(Listing continued, text begins on page 4.)*

```
* DESC: Grab a point and move it. Return 1 when key is pressed, else return 0.
*****************************************************************/
int
movePoint(LINE_LIST *lineList)
{
    int     stuckx, stucky, movex,movey;
    int     oldx, oldy;
    int     input;
    int     line, point;
    /* save the current mode */
    short   old_mode = _getwritemode();
    do {
        /* keep getting input until we have a mouse button */
        while (!(input = mousePos (&movex, &movey)));
        if (input & KEYPRESS) {
            _setwritemode(old_mode);
            return 1;
        }
        if (!findPoint(lineList, &line, &point, movex, movey)) {
            _setwritemode(old_mode);
            return 0;
        }
        /* establish fixed end point */
        stuckx = lineList->line[line].p[1-point].x;
        stucky = lineList->line[line].p[1-point].y;
        oldx=movex, oldy=movey;
        hideMouse();
        /* erase the old line */
        xorLine (stuckx, stucky,
                 lineList->line[line].p[point].x,
                 lineList->line[line].p[point].y);
        /* and prime the pump with the new line */
        xorLine (stuckx, stucky, oldx, oldy);
        showMouse();
        while (input = mousePos (&movex, &movey)) {
            /* rubber band a line while the mouse is dragged */
            if (movex != oldx || movey != oldy) {
                hideMouse();
                xorLine (stuckx, stucky, oldx, oldy);
                xorLine (stuckx, stucky, movex, movey);
                showMouse();
                oldx=movex, oldy=movey;
            }
        }
    } while (stuckx == movex && stucky == movey);
    lineList->line[line].p[point].x = movex;
    lineList->line[line].p[point].y = movey;
    _setwritemode(old_mode);        /* get out of XOR mode */
    return (0);
}
/*****************************************************************
* FUNC: void    createLines(PICTURE *pic, LINE_LIST *lineList)
* DESC: create a list of line segments for a picture
*****************************************************************/
void
createLines(PICTURE *pic, LINE_LIST *lineList)
{
    setupScreen(pic, 0);    /* set for enter prompt */
    initMouse();
    showMouse();
    for (lineList->number = 0;lineList->number < MAX_LINES;
                                            lineList->number++) {
        if (getLine(&lineList->line[lineList->number].p[0].x,
            &lineList->line[lineList->number].p[0].y,
            &lineList->line[lineList->number].p[1].x,
            &lineList->line[lineList->number].p[1].y))
            break;
    }
    hideMouse();
}
/*****************************************************************
* FUNC: void    editLines(PICTURE *pic, LINE_LIST *lineList)
* DESC: move around some existing lines
*****************************************************************/
void
editLines(PICTURE *pic, LINE_LIST *lineList)
{
    int segment;
    setupScreen(pic, 1);    /* set for edit prompt */
    initMouse();
    for (segment = 0; segment < lineList->number; segment++) {
       xorLine(lineList->line[segment].p[0].x, lineList->line[segment].p[0].y,
               lineList->line[segment].p[1].x, lineList->line[segment].p[1].y);
    }
    showMouse();
    /* move the endpoints around */
    while(!movePoint(lineList));
    hideMouse();
}
/*****************************************************************
* FUNC: void    setupScreen(PICTURE *pic, int editFlag)
* DESC: Print a message introducing the screen, wait for input,
*       then set the graphics mode and display the screen.
*****************************************************************/
void
setupScreen(PICTURE *pic, int editFlag)
{
    static char *editMess[2] = {"enter", "edit"};
    static char *targMess[2] = {"source", "target"};
    setTextMode();
    _settextposition(VTAB, HTAB);
    printf("When you are ready to %s the control lines", editMess[editFlag]);
    _settextposition(VTAB+2, HTAB);
    printf("for the %s image, press any key.", targMess[TargFlag]);
    waitForKey();
    setGraphicsMode();
    displayPicture(pic);
}
```

Listing Two

```
/*****************************************************************
* FILE: define.h
* DESC: These are the main defines for dissolve, warp,  morph, load and fix.
* HISTORY:  Created  1/11/1993   LAST CHANGED:  5/ 6/1993
*   Copyright (c) 1993 by Scott Anderson
*****************************************************************/

/* ----------------------DEFINES---------------------------- */
#define ON              1
#define OFF             0
#define MAX_TWEENS      99      /* Maximum tweens (2 digits) */
/* minus 2 digit tween# appended to end */
#define MAX_NAME_SIZE   (8-2)
#define HEADER_LEN      128     /* PCX header length */
/* Number of colors in the palette */
#define COLORS          256
/* bytes in palette (COLORS*3) */
#define PALETTE_SIZE    (3*COLORS)
/* Maximum number of morphing lines */
#define MAX_LINES       32
/* max number of pixels wide we handle */
#define MAX_WIDE        320
/* max number of pixels tall we handle */
#define MAX_TALL        200
/* Size of screen buffer */
#define MAX_BYTES       (MAX_WIDE*(long) MAX_TALL)
/* Number of components per color (RGB) */
#define COMPS           3
/* largest color component value */
#define MAX_COMP        32
/* the midpoint of the colors - for gray */
#define MID_COMP        (MAX_COMP/2)
/* enough to handle about 10 different palettes */
#define MAX_FREQ        1023
#define MAX_FILES       10
/* length of a file name including directory */
#define MAX_PATHLEN             80
#define ENTER           13  /* Keyboard values */
#define ESC         27
#define HTAB            18  /* Position for text messages */
#define VTAB            8
/* The mouse button & keyboard constants */
#define NO_BUTTON       0
#define LEFT_BUTTON     1
#define RIGHT_BUTTON            2
#define KEYPRESS        4
/* the square of min dist for grabbing pt */
#define GRAB_DISTANCE   25
/* Some of the graphics colors */
#define BLACK           0
#define WHITE           255
#define EXT_PCX         ".PCX"  /* pcx file extension */
/* primary line file holder extension */
#define EXT_LINE1       ".LN1"
#define EXT_LINE2       ".LN2"  /* aux file for warp lines */
#define ERROR           -1  /* General-purpose error code */

typedef enum {
    NO_ERROR,               /* first entry means everything is ok */
    MEMORY_ERR,             /* Not enough memory */
    READ_OPEN_ERR,          /* Couldn't open file for reading */
    READ_ERR,               /* Trouble reading the file */
    WRITE_OPEN_ERR,         /* Couldn't open the file for writing */
    WRITE_ERR,              /* Couldn't write the file */
    MOUSE_ERR,              /* No mouse driver found */
    WRONG_PCX_FILE,         /* PCX file format not supported yet */
    READ_CONTENTS_ERR       /* error in .LN file */
}
ERR;
/* ----------------------MACROS----------------------------- */
#define MIN(a,b)        (((a)<(b)) ? (a) : (b))
#define PIXEL(p,x,y)    (p->pixmap[y * (long) p->wide + x])
#define SQUARE(x)       (((long) x)*(x))
/* ----------------------TYPEDEFS--------------------------- */
typedef struct {
    int x,y;        /* the screen coordinates of the point */
}
POINT;
typedef struct {
    POINT p[2];
}
LINE_SEGMENT;
typedef struct {
    int number;     /* number of segments to follow */
    LINE_SEGMENT line[MAX_LINES];
    char *filename; /* name of file holding the line list */
}
LINE_LIST;
typedef struct {
    POINT p[2];             /* the endpoints */
    int delta_x, delta_y;   /* x & y displacement */
    float length;           /* the precalculated length of the line */
    long length_square;     /* the length squared */
}
LINE;
typedef struct {
    /* red, green, and blue color components */
    unsigned char r, g, b;
}
COLOR;
typedef struct {
    COLOR c[COLORS];        /* a 256 entry palette */
}
PALETTE;
typedef struct {
    int xmin, ymin;         /* the upper left corner */
    int xmax, ymax;         /* the lower right corner */
    int wide, tall;         /* the width and height */
    int pal_id;         /* an ID number for each palette */
    PALETTE pal;            /* the actual palette is here */
    unsigned char far *pixmap;  /* a pointer to the pixel map */
}
PICTURE;
typedef struct linko {
    struct linko *next;
    char        *str;
}
LINKED_LIST;
/* ----------------------PROTOTYPES------------------------- */
/**** file handling routines ****/
extern PICTURE  *loadPicture(char *filename);
extern int      loadPalette(FILE *fp, PALETTE *palette);
extern int      getBlock (unsigned char *byte, int *count, FILE *fp);
extern int      mustRead(FILE *fp, char *buf, int n);
extern int      saveScreen(PALETTE *pal);
extern int      putBlock(unsigned char num, unsigned char color, FILE *fp);
extern int      writeByte(unsigned char *byte, FILE *fp);
/**** screen and color routines ****/
extern int      defaultPalette(PALETTE *palette);
extern int      setPalette(PALETTE *palette);
extern int      displayPicture(PICTURE *picture);
extern int      displayNoPal(PICTURE *picture);
extern int      freePicture(PICTURE *pic);
/**** mouse routines ****/
extern int      initMouse();
extern int      hideMouse();
extern int      showMouse();
extern int      mousePos(int *x, int *y);
/**** general purpose routines ****/
extern int      clip(int num, int min, int max);
extern int      quitCheck();
extern void     quit(int err, char *name);
extern int      wait(int count);
extern int      waitForKey();
extern char     lineAsk(char *name);
/* ----------------------GLOBAL DATA------------------------ */
extern int      TargFlag;
extern int      Key;
```

Listing Three

```
/*****************************************************************
* FILE: morph.c
* DESC: Create a metamorphosing sequence between two given images. This
*     program lets you specify two files to morph, then prompts you for
*     control lines. It uses the lines to warp the underlying images a step
*     at a time, combine them, and optionally save them as numbered PCX files.
*HISTORY:  Created  1/13/1993    LAST CHANGED:  5/ 6/1993
*   Copyright (c) 1993 by Scott Anderson
*****************************************************************/

/* ----------------------INCLUDES--------------------------- */
#include <conio.h>
#include <stdio.h>
#include <io.h>
#include <math.h>
#include <graph.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include "define.h"

/* ----------------------DEFINES---------------------------- */
#define MORPH_TWEENS    1

/* ----------------------PROTOTYPES------------------------- */
int     tweenMorph(PICTURE *src, PICTURE *dst);

/* ----------------------EXTERNALS-------------------------- */
/**** color routines ****/
extern int  closestColor(int r, int g, int b, PALETTE *palPtr);
extern void collapseColors(PALETTE *palPtr);
/**** line routines ****/
extern int  setLength(LINE *line);
extern int  sumLines(PICTURE *picture, COLOR *color,
                LINE *origline, POINT *warp, LINE *warpline);
/**** io routines ****/
extern LINE_LIST    *loadLines(char *filename, char *extension);
extern void          saveLines(char *filename,
                LINE_LIST *lineList, char *extension);

/***** variables used to compute intermediate images ****/
/* number of colors in tweened image before reduction*/
extern int  Ncolors;
/* r, g, b frequency counter array */
extern unsigned int far Freq[MAX_COMP][MAX_COMP][MAX_COMP];
/* tweened images red, grn, and blu components*/
extern unsigned char far Red[MAX_WIDE][MAX_TALL];
extern unsigned char far Grn[MAX_WIDE][MAX_TALL];
extern unsigned char far Blu[MAX_WIDE][MAX_TALL];
extern PALETTE TweenPal;            /* resulting palette */

/**** other variables ****/
extern char     *OutFilename;
/* set from last picture loaded */
extern int      Xmin, Ymin, Xmax, Ymax;
/* ID of palette currently being displayed */
extern int      CurrentPal;

/* ----------------------GLOBAL DATA------------------------ */
PICTURE *Src;       /* source & destination picture pointers */
PICTURE *Dst;
LINE    SrcLine[MAX_LINES];
LINE    DstLine[MAX_LINES];
int     Tweens;
int     NumLines;

/*****************************************************************
* FUNC: main (int argc, char *argv[])
* DESC: Read in a filename to load
*****************************************************************/
main (int argc, char *argv[])
{
    int     segment;
    LINE_LIST *lineSrcList;
    LINE_LIST *lineDstList;
    char    answer;
    /* load the pcx file if one is given */
    if ((3 > argc) || (argc > 5)) {
        printf("Usage: morph <source><dest> [<steps> [<output>]]\n\n");
        printf("Where: <source> is the source PCX filename\n");
        printf("       <dest>   is the destination filename\n");
        printf("       <steps>  is the optional sequence size\n");
        printf("                (the max is %d, the default is %d)\n",
                MAX_TWEENS, MORPH_TWEENS+2);
        printf("       <output> is the optional output filename\n");
        printf("                (defaults to no output)\n\n");
        printf("Note:  The output filename can be at most %d
                                characters long.\n", MAX_NAME_SIZE);
        printf("       The PCX extension is added automatically,
                                                    so don't\n");
        printf("       include it in the filename.\n");
        printf("       Morph only accepts PCX files with %d X %d
                                resolution\n", MAX_WIDE, MAX_TALL);
        printf("       and %d colors.\n", COLORS);
        exit(0);
    }
    if (argc > 3) {
        /* subtract two from the series count to get the tweens
         * since the starting and ending frame are included. */
        Tweens = clip (atoi(argv[3]) - 2, 1, MAX_TWEENS);
        if (argc > 4)
            OutFilename = argv[4];
    }
    else
        Tweens = MORPH_TWEENS;
    printf("Loading the file %s\n", argv[1]);
    Src = loadPicture(argv[1]);
    if (Src == NULL)
        quit(MEMORY_ERR, "");
    printf("Loading the file %s\n", argv[2]);
    Dst = loadPicture(argv[2]);
    if (Dst == NULL)
        quit(MEMORY_ERR, "");
    lineSrcList = loadLines(argv[1], EXT_LINE1);
    if (lineSrcList->number != 0) {
        if (lineAsk(argv[1]) == 'N')
            createLines(Src, lineSrcList);
        else
            editLines(Src, lineSrcList);
    }
    else
        createLines(Src, lineSrcList);

    TargFlag = 1;   /* For the screen intro message */
    NumLines = lineSrcList->number;
    if (NumLines) {
        lineDstList = loadLines(argv[2], EXT_LINE1);
            /* inconsistent warp target*/
        if (lineDstList->number !=  NumLines)
            lineDstList->number = 0;
        if (lineDstList->number) {  /* ask what he wants to do */
            if (lineAsk(argv[2]) == 'N')
                lineDstList->number = 0;
        }
        if (lineDstList->number == 0) { /* create a warp target */
            /* copy the source lines */
            lineDstList->number = NumLines;
            for (segment = 0; segment < NumLines; segment++)
                lineDstList->line[segment] = lineSrcList->line[segment];
        }
        editLines(Dst, lineDstList);
        saveLines(argv[1], lineSrcList, EXT_LINE1);
        saveLines(argv[2], lineDstList, EXT_LINE1);
        beep();
        for (segment = 0; segment < NumLines; segment++) {
            DstLine[segment].p[0]=lineDstList->line[segment].p[0];
            DstLine[segment].p[1]=lineDstList->line[segment].p[1];
            setLength(&DstLine[segment]);
            SrcLine[segment].p[0]=lineSrcList->line[segment].p[0];
            SrcLine[segment].p[1]=lineSrcList->line[segment].p[1];
            setLength(&SrcLine[segment]);
        }
    }
    tweenMorph(Src, Dst);
    setTextMode();
}
/*****************************************************************
* FUNC: int tweenMorph(PICTURE *src, PICTURE *dst)
* DESC: calculate a pixel to plot, from the warping function
*****************************************************************/
#define TOTAL_WEIGHT        (100)   /* Good for up to 99 tweens */
tweenMorph(PICTURE *src, PICTURE *dst)
{
    int color;
    POINT warp;
    int x,y;
    COLOR scolor, dcolor;
    LINE warpLine[MAX_LINES];
    int t, i, p;
    int r, g, b;
    unsigned int srcweight, srcpaletteindex;
    unsigned int dstweight, dstpaletteindex;
    displayPicture(src);
    saveScreen(&src->pal);
    /* src is on screen, now tween to the target */
    for (t = 1; t <= Tweens; t++) {
        /* Tween the lines used to warp the images */
        for (i = 0; i < NumLines; i++) {
```

(continued on page 78)

Listing Three *(Listing continued, text begins on page 4.)*

```
        for (p = 0; p < 2; p++) {
            warpLine[i].p[p].x = SrcLine[i].p[p].x +
              ((DstLine[i].p[p].x - SrcLine[i].p[p].x) * t)
              /(Tweens+1);
            warpLine[i].p[p].y = SrcLine[i].p[p].y +
              ((DstLine[i].p[p].y - SrcLine[i].p[p].y) * t)
              /(Tweens+1);
        }
        setLength(&warpLine[i]);
    }
    dstweight = t * TOTAL_WEIGHT / (Tweens+1);
    srcweight = TOTAL_WEIGHT - dstweight;
    /* Zero out the buffers */
    initFreq();
    /* set background to black */
    _fmemset(Red, 0, sizeof Red);
    _fmemset(Grn, 0, sizeof Grn);
    _fmemset(Blu, 0, sizeof Blu);
    /* Go through the screen and get warped source pixels */
    for (warp.y = Ymin; warp.y <= Ymax; warp.y++)   {
        if (quitCheck())
            quit(0, "");
        for (warp.x = Xmin; warp.x <= Xmax; warp.x++)   {
             sumLines(src, &scolor, SrcLine, &warp, warpLine);
             sumLines(dst, &dcolor, DstLine, &warp, warpLine);
             r = (scolor.r * srcweight + dcolor.r * dstweight)
                      / TOTAL_WEIGHT;
             g = (scolor.g * srcweight + dcolor.g * dstweight)
                      / TOTAL_WEIGHT;
             b = (scolor.b * srcweight + dcolor.b * dstweight)
                      / TOTAL_WEIGHT;
             if (Freq[r][g][b] == 0)    /* A new color */
                Ncolors++;
            /* Keep it to one byte */
            if (Freq[r][g][b] < MAX_FREQ)
                Freq[r][g][b]++;
            /* put RGB components into temporary buffer */
            Red[warp.x][warp.y] = r;
            Grn[warp.x][warp.y] = g;
            Blu[warp.x][warp.y] = b;
        }
    }
    collapseColors(&TweenPal);
    setPalette(&TweenPal);
    for (y = Ymin; y <= Ymax; y++)  {
        if (quitCheck())
            quit(0, "");
        for (x = Xmin; x <= Xmax; x++)  {
            color = closestColor(   Red[x][y], Grn[x][y],
                                    Blu[x][y], &TweenPal);
            _setcolor (color);
            _setpixel (x, y);
        }
    }
    /* no output file name on command line */
    if (!OutFilename) {
        beep();
        waitForKey();   /* so pause to enjoy the pictures */
    }
    else
        saveScreen(&TweenPal);
}
if (OutFilename) {  /* save the last pic in this series */
    CurrentPal = 0;         /* force a new palette */
    displayPicture(dst);
    saveScreen(&dst->pal);
    }
}
```

Listing Four

```
/*********************************************************************
* FILE: load.c
* DESC: This program loads a PCX file or a list of them. It crams as
*       many into memory as it can, then it flips quickly through them.
* HISTORY:  Created  1/13/1993    LAST CHANGED:  3/20/1993
*   Copyright (c) 1993 by Scott Anderson
*********************************************************************/
/* ----------------------INCLUDES----------------------------- */
#include <conio.h>
#include <stdio.h>
#include <io.h>
#include <math.h>
#include <graph.h>
#include <string.h>
#include "define.h"
/* ----------------------EXTERNALS---------------------------- */
/* External functions */
extern int  quitCheck();
extern LINKED_LIST  *rootSequence(int argc, char *argv[]);
/* External variables */
extern int  Wait;
extern int  Key;
extern int  EndWait;
/* ----------------------GLOBAL DATA---------------------------- */
PICTURE *Src[MAX_FILES];            /* source picture pointer */
/*********************************************************************
* FUNC: main (int argc, char *argv[])
* DESC: Display the file or sequence passed on the command line. Read in as
*       many files as will fit in memory, then display them in a loop.
*********************************************************************/
main (int argc, char *argv[])
{
    int file, fileNum;
    int direction;
    int i;
    LINKED_LIST *pcxList;
    LINKED_LIST *pcxListHead;
    if (argc == 1) {
        printf("Usage: load <name>\n\n");
        printf("Where: <name> is the root name of a sequence\n");
        exit(23);
    }
    setGraphicsMode();
    file = 1;
    pcxListHead =  rootSequence(argc, argv);
    for (pcxList=pcxListHead; pcxList; pcxList = pcxList->next) {
        Src[file] = loadPicture(pcxList->str);
        if (Src[file] == NULL)
            break;
        displayPicture(Src[file++]);
    }
    fileNum = file - 1;
    if (fileNum == 1)   /* there's only one file */
        waitForKey();   /* so wait for the user to quit */
    else if (fileNum > 1) {
        file = 1;
        direction = 1;
        while (!(quitCheck())) {
            if ((file += direction) >= fileNum)
                direction = -1;
            if (file <= 1)
                direction = 1;
            displayPicture(Src[file]);
            if (EndWait && (file == 1 || file == fileNum))
                wait(Wait);
        }
    }
    /* Reset to original mode, then quit */
    setTextMode();
}
```

Listing Five

```
/*********************************************************************
* FILE: fix.c
* DESC: This program inputs a list of pictures, creates a best
*       fit palette, remaps the pictures, and writes them out.
* HISTORY:  Created  1/13/1993   LAST CHANGED: 3/10/1993
*   Copyright (c) 1993 by Scott Anderson
*********************************************************************/
/* ----------------------INCLUDES----------------------------- */
#include <conio.h>
#include <stdio.h>
#include <io.h>
#include <math.h>
#include <graph.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include "define.h"
/* ----------------------EXTERNALS---------------------------- */
extern LINKED_LIST  *rootSequence(int argc, char *argv[]);
/**** color routines ****/
extern int      closestColor(int r, int g, int b, PALETTE *pal);
extern void     collapseColors(PALETTE *palPtr);
extern int      mergePalette(PICTURE *pic);
extern int      remapPicture(PICTURE *picPtr, PALETTE *palPtr);
/**** line routines ****/
extern int      getLine(int *argx1, int *argy1, int *argx2, int *argy2);
extern int      movePoint();
extern int      setLength(LINE *line);
/**** other variables ****/
extern char     *OutFilename;
/* set from last picture loaded */
extern int      Xmin, Ymin, Xmax, Ymax;
/* ----------------------GLOBAL DATA---------------------------- */
PICTURE *Src;       /* source & destination picture pointers */
/***** variables used to compute intermediate images ****/
/* number of colors in tweened image before reduction*/
extern int  Ncolors;
/* r, g, b frequency counter array */
extern unsigned int far Freq[MAX_COMP][MAX_COMP][MAX_COMP];
/* tweened images red, grn, and blu components*/
extern unsigned char far Red[MAX_WIDE][MAX_TALL];
extern unsigned char far Grn[MAX_WIDE][MAX_TALL];
extern unsigned char far Blu[MAX_WIDE][MAX_TALL];
extern PALETTE TweenPal;            /* resulting palette */
/*********************************************************************
* FUNC: main (int argc, char *argv[])
* DESC: Read in a list of filenames to load, change their palettes
*       to the best-fit palette, and write them out.
*********************************************************************/
main (int argc, char *argv[])
{
    int     file;
    LINKED_LIST *pcxList, *pcxListHead;
    /* load the pcx file if one is given */
    if (argc < 3) {
        printf("Usage: fix <infile> <outfile)\n\n");
        printf("Where: <infile>  is the input sequence name\n");
        printf("       <outfile) is the output sequence name\n");
        exit(0);
    }
    OutFilename = argv[argc-1];
    initFreq();
    pcxListHead =  rootSequence(argc-1, argv);
    for (pcxList = pcxListHead; pcxList; pcxList=pcxList->next) {
        printf("Loading the file %s\n", pcxList->str);
        Src = loadPicture(pcxList->str);
        if (Src == NULL)
            quit(MEMORY_ERR, "");
        mergePalette(Src);
        freePicture(Src);
    }
    collapseColors(&TweenPal);
    setGraphicsMode();
    setPalette(&TweenPal);
    for (pcxList = pcxListHead; pcxList; pcxList=pcxList->next) {
        Src = loadPicture(pcxList->str);
        if (Src == NULL)
            quit(MEMORY_ERR, "");
        remapPicture(Src, &TweenPal);
        displayNoPal(Src);
        saveScreen(&TweenPal);
        freePicture(Src);
    }
    setTextMode();
}
```

Listing Six

```
/*********************************************************************
* FILE: color.c
* DESC: This file contains the color routines used by morph, dissolve and fix.
* HISTORY:  Created  3/18/1993  LAST CHANGED: 5/ 6/1993
*   Copyright (c) 1992 by Scott Anderson
*********************************************************************/
#include <stdio.h>
#include <memory.h>
#include "define.h"
/* ----------------------DEFINES------------------------------ */
/* ----------------------TYPEDEFS/STRUCTS--------------------- */
/* ----------------------PROTOTYPES--------------------------- */
int     closestColor(int r, int g, int b, PALETTE *palPtr);
void    collapseColors(PALETTE *palPtr);
int     mergePalette(PICTURE *pic);
int     remapPicture(PICTURE *picPtr, PALETTE *palPtr);
int     initFreq();
/* ----------------------EXTERNALS---------------------------- */
/* set from last picture loaded */
extern int      Xmin, Ymin, Xmax, Ymax;
/* ----------------------GLOBAL DATA---------------------------- */
/* number of colors in tweened image before reduction*/
int     Ncolors;
/* r, g, b frequency counter array */
unsigned int far Freq[MAX_COMP][MAX_COMP][MAX_COMP];
/* tweened images red, grn, and blu components*/
unsigned char far Red[MAX_WIDE][MAX_TALL];
unsigned char far Grn[MAX_WIDE][MAX_TALL];
unsigned char far Blu[MAX_WIDE][MAX_TALL];
PALETTE TweenPal;           /* resulting palette */
/*********************************************************************
* FUNC: void    collapseColors(PALETTE *palPtr)
* DESC: Collapse the colors in the Freq table until
*       Ncolors < COLORS, then put it in the given color palette.
*********************************************************************/
void
collapseColors(PALETTE *palPtr)
{
    int freqCutoff;
    int r, g, b;
    int index;
    int ncolors;
    static int freqCount[MAX_FREQ+1];
    memset(freqCount, 0, sizeof freqCount);
    for (r = 0; r < MAX_COMP; r++)
        for (g = 0; g < MAX_COMP; g++)
            for (b = 0; b < MAX_COMP; b++)
                freqCount[Freq[r][g][b]]++;
    ncolors = 0;
    for (freqCutoff = COLORS-1; freqCutoff > 1; freqCutoff--) {
        ncolors += freqCount[freqCutoff];
        if (ncolors > COLORS) break;
    }
    /* Collapse color space to 256 colors */
    r = g = b = 0;
    while (Ncolors >= COLORS) {
        for (; r < MAX_COMP; r++, g=0) {
            for (; g < MAX_COMP; g++, b=0) {
                for (; b < MAX_COMP; b++) {
                    if (Freq[r][g][b] && Freq[r][g][b]
                        <= freqCutoff)
                    goto castOut;   /* the ultimate no no */
                    }
            }
        }
        r = g = b = 0;
        freqCutoff++;
        continue;
    castOut:
        Freq[r][g][b] = 0;  /* just remove this low freq color */
        Ncolors--;
    }
    /* build a palette out of all the remaining non zero freq's */
    index = 0;
    for (r = 0; r < MAX_COMP; r++)
        for (g = 0; g < MAX_COMP; g++)
            for (b = 0; b < MAX_COMP; b++)
                /* we have a color we need to map */
                if (Freq[r][g][b]) {
                    palPtr->c[index].r = r;
                    palPtr->c[index].g = g;
                    palPtr->c[index].b = b;
                    /* remember index in palette */
                    Freq[r][g][b] = index;
                    index++;
                }
}
```

(continued on page 80)

MARKETPLACE

CD-ROM

CDROM-NOW!™ for Windows NT

Save $$$!

CDROM NOW! is a high performance Windows NT CDROM driver optimized for support of inexpensive non-SCSI CDROM drives. Maximize NT performance on Mitsumi LU002, LU005S. FX001, FX001D, DAK, Sony CR535, Panasonic CR521 or 522 drives.

Features: Easy installation, ISO data access, CD audio, XA support, Kodak Photo CD support, Networked CD sharing and more!

Order now! Only $49.95 plus $5(S&H)

Limited Time Special Offer: A FREE, brand new graphic CD Title with purchase! A $39.95 value!

Media Synergy Inc.

7Carlton Street, Suite 1404, Toronto, Canada M5B 2M3
TEL: (416) 593-0647 FAX: (416) 593-8772

Site Licenses and Consultants Volume Paks available

CIRCLE NO. 1110 ON READER SERVICE CARD

MARKETING

Get Your Multimedia Software To The Right People!

If you are selling software to multimedia developers or are a multimedia software developer yourself, you need to be in **SoftGuide '94**. Advertise your multimedia software library, algorithm, API, application development tool, or your multimedia expertise at unheard of rates. **SoftGuide** is an attractive alternative to expensive ad space and focuses exclusively on the software development community. Special incentives apply for first time advertisers.

Software Guides
1800 Wyatt Drive, Suite 10, Dept. C, Santa Clara, CA 95054
800-807-7711 or 408-496-5741

CIRCLE NO. 1111 ON READER SERVICE CARD

RECRUITMENT

FREE Bulletin Board

413-549-8136 (8-N-1)

Direct dial. Baud rates to 14,400.

If you have access to a modem, we invite you to navigate our listings of hundreds of today's hottest search assignments nationwide, plus forums and career guides.

Software and IS jobs

Allen Davis & Associates, National Technical Search
PO Box 2007 • Amherst, MA 01004-2007
tel: 413-549-7440 • fax: 413-549-7542
Call or write in confidence. Client companies pay all fees.

CIRCLE NO. 1112 ON READER SERVICE CARD

Programming & Systems Consultants

PROVIDES
Management Consulting
Technical Education
Contract Resources
and
Turn-key System Development
Specializing in
Client-Server & Open System Technologies

OOD/OOP • C++
UNIX • NeXT
MS-Windows
SDK
X-Windows
Motif
Oracle-Sybase
SQL Server
Visual Basic
Visual C++

For immediate interview, call: (314) 878-4777
Resumes may be faxed: (314) 878-4775
or mailed to: 12977 N Outer 40 Drive • Suite 310 • St. Louis • MO 63141

CIRCLE NO. 1113 ON READER SERVICE CARD

RECRUITMENT

Ready For New Challenges? Positions Coast-To-Coast

C, C++, OOP, GUI, Motif, Sybase, Windows, LISP, Event Driven Programming, Compiler Development, ML, Device Driver Development, MAC, CD-ROM, Multimedia...

Hey! How far do we have to go with this list?

Call or fax us at:

THE SHAY GROUP

P.O. Box 398 Amherst, MA 01004-0398
Voice: 413/256-1030 Fax: 413/256-8582

Fee Paid By Employer

CIRCLE NO. 1114 ON READER SERVICE CARD

Some people donate money to keep the air clean. Jeff Angus wrote 2000 lines of code.

© 1991 CompuMentor

Jeff Angus has regularly donated money to the Natural Resources Defense Council. Now he also supports them in a different and maybe even more valuable way. By donating his skills to modify a dBASE program that generates customized reports about toxic substances.

If you'd like to put your own special skills to good use, call 1-800-659-3579. And we'll match you with a local non-profit organization whose needs are compatible with your area of expertise. Call us now, and we'll show you a great way to make a real difference in the world. By doing what you do best.

CompuMentor®

People helping computers help people.

89 Stillman Street, San Francisco, CA 94107

Listing Six *(Listing continued, text begins on page 4.)*

```
/*********************************************************************
* FUNC: int closestColor(int r, int g, int b, PALETTE *palPtr)
* DESC: return the palette index of the color closest to rgb.
*********************************************************************/
int
closestColor(int r, int g, int b, PALETTE *palPtr)
{
    int index;
    int distance;
    int min_distance = 3200;    /* a big number */
    int min_index;
    /* The value in Freq is now the index into the color table */
    if (Freq[r][g][b]) return Freq[r][g][b];
    /* If zero, search for the closest color */
    for (index = 1; index < Ncolors; index++) {
        /* this is really the distance squared, but it works */
        distance =  SQUARE (r - palPtr->c[index].r) +
                    SQUARE (g - palPtr->c[index].g) +
                    SQUARE (b - palPtr->c[index].b);
        if (distance < min_distance) {
            min_distance = distance;
            min_index = index;
            if (distance <= 2) break;   /* close enough! */
        }
    }
    /* New index - for future reference */
    Freq[r][g][b] = min_index;
    return min_index;
}
/*********************************************************************
* FUNC: int mergePalette(PICTURE *picPtr)
* DESC: Merge a palette into Freq count table.
*********************************************************************/
int
mergePalette(PICTURE *picPtr)
{
    int     r, g, b;
    unsigned int    pos;
    unsigned char   index;
    PALETTE *palPtr = &picPtr->pal;
    unsigned char far *bufPtr = picPtr->pixmap;
    for (pos = 0; pos < MAX_BYTES; pos++) {
        index = *bufPtr++;
        r = palPtr->c[index].r;
        g = palPtr->c[index].g;
        b = palPtr->c[index].b;
        if (Freq[r][g][b] == 0)     /* A new color */
            Ncolors++;
        if (Freq[r][g][b] < MAX_FREQ)   /* Keep it managable */
            Freq[r][g][b]++;
    }
}
/*********************************************************************
* FUNC: int remapPicture(PICTURE *picPtr, PALETTE *palPtr)
* DESC: Remap a picture with a different palette.
*********************************************************************/
int
remapPicture(PICTURE *picPtr, PALETTE *palPtr)
{
    int     x, y;
    int     index;
    int     r, g, b;
    unsigned int    pos;
    unsigned char   lookup[COLORS];
    unsigned char far *bufPtr;
    /* Create the cross-reference lookup table */
    for (index = 0; index < COLORS; index++) {
        r = picPtr->pal.c[index].r;
        g = picPtr->pal.c[index].g;
        b = picPtr->pal.c[index].b;
        lookup[index] = closestColor(r, g, b, palPtr);
    }
    /* Save the new palette in the picture's palette */
    for (index = 0; index < COLORS; index++) {
        picPtr->pal.c[index].r = palPtr->c[index].r;
        picPtr->pal.c[index].g = palPtr->c[index].g;
        picPtr->pal.c[index].b = palPtr->c[index].b;
    }
    /* Remap the individual pixels to point to the new colors */
    for (bufPtr = picPtr->pixmap, pos = 0; pos < MAX_BYTES;
                        bufPtr++, pos++)
        *bufPtr = lookup[*bufPtr];
}
/*********************************************************************
* FUNC: int initFreq()
* DESC: zero out the frequency color space table
*********************************************************************/
int
initFreq()
{
    int bytes = (sizeof Freq) / 2;
    _fmemset(Freq, 0, bytes);
    /* divide because of element size */
    _fmemset(Freq+(bytes/sizeof *Freq), 0, bytes);
    /* Guarantee a black color */
    Freq[0][0][0] = MAX_FREQ;
    /* a grey color */
    Freq[MID_COMP-1][MID_COMP-1][MID_COMP-1] = MAX_FREQ;
    /* and a white color */
    Freq[(long)MAX_COMP-1][MAX_COMP-1][MAX_COMP-1] = MAX_FREQ;
    Ncolors = 3;
}
```

End Listings

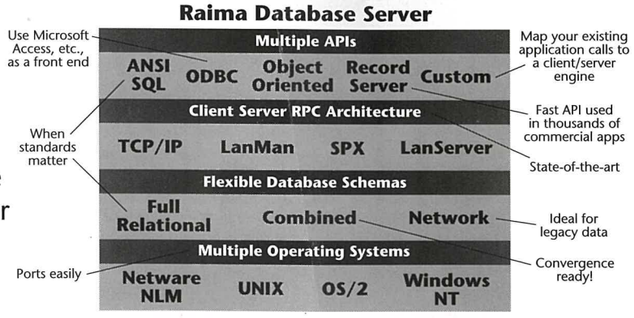
DEVELOPERS:

Have you been asked to produce a fast, open, client/server application that is inexpensive to deploy and differentiates your company?

Then we have a suggestion...

use our engine.

The new **Raima Database Server** is a product designed for developers who value flexibility along with speed and support for standards. Your application design is not constrained by the database interface. Why? With RDS, we begin with a multithreaded client/server architecture coupled to a high performance SQL processor/optimizer. Then we add support for C++ objects, a fast record-oriented API, and custom extension modules featuring remote procedure calls from your client application to your server-hosted user defined functions. Further, your database schema can be fully relational, network, or a combination of the two. It's all there for you.



Supports stored procedures, triggers, and referential integrity!

Join the thousands of developers, from small vertical market ISVs to industry giants like Hewlett Packard, who use our technology to set their applications and products apart.

Call **1-800-DB-RAIMA** for more information.

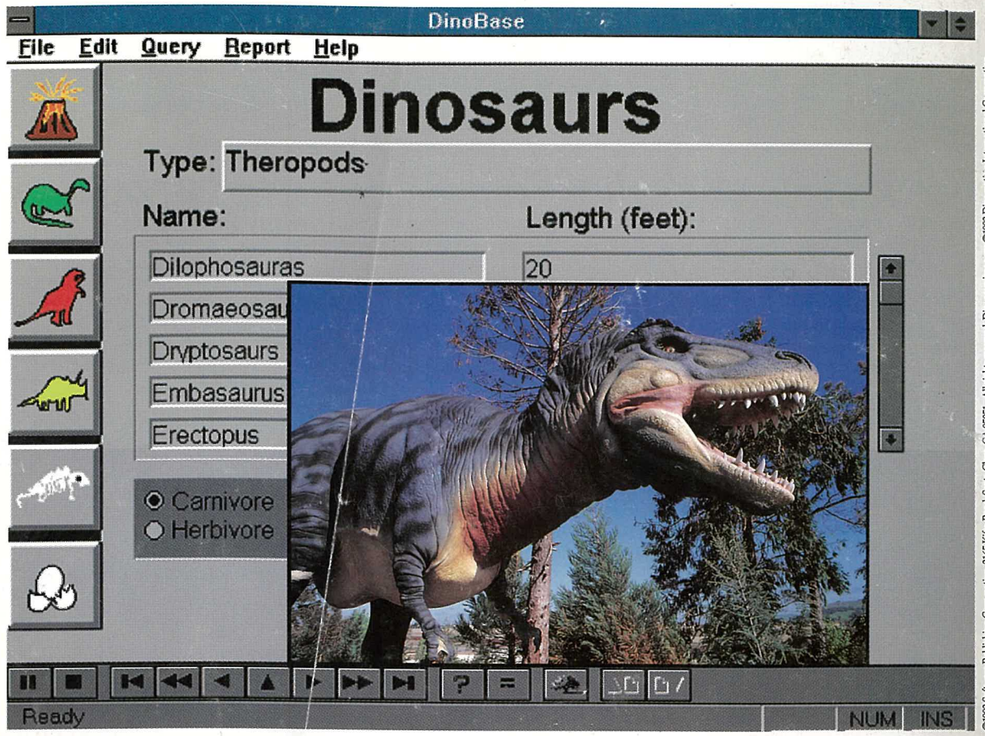
RAIMA™

DD133RDS

Raima Corporation, 1605 NW Sammamish Rd., Issaquah, WA 98027 (206) 557-0200 Fax: (206) 557-5200

© 1993 Raima Corporation. Raima Database Server is a trademark of Raima Corporation. Raima is a registered trademark of Raima Corporation. Other computer and software names are trademarks or registered trademarks of their respective holders.

Raima Benelux - 31 (2159) 44738 Raima Deutschland 49 (7022) 9256 44 Raima España 34 (3) 485 13 04 Raima Italia - 39 (49) 829 1285
Raima UK Ltd. - 44 (276) 685 761 Rhône-Alpes - 41 (21) 825 4800 France - 33 (1) 46 48 61 02

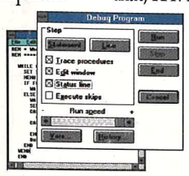


©1993 Software Publishing Corporation, 3165 Kifer Road, Santa Clara, CA 95051. All rights reserved. Dinosaur imagery ©1993 Dinamation International Corporation.

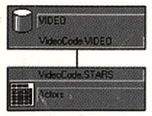
You'll be amazed at what Superbase can bring to life.

Superbase® gives you the power to add more life to virtually any database application you can imagine. With graphical images, full-motion video and audio, Superbase gets you from idea to roaringly successful business application in minutes.

Much of the power behind the punch of Superbase applications is our visual tools. Our Relational Advanced Visual Environment (RAVE) shows you visually what you're programming on-screen, so you can build fast applications that literally leap with life in no time.

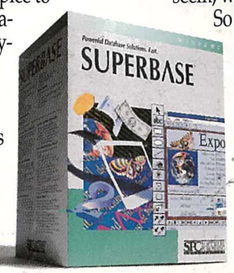


Build custom menus, icon bars, dialogs, forms and reports—even customize Superbase—just by pointing-and-clicking, dragging-and-dropping. RAVE also links databases easily, shows you the results visually, and lets you custom-configure Superbase's feature set for different levels of users.

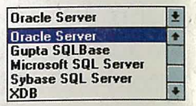


You'll feel right at home with Super Basic Language (SBL), which is similar to Visual Basic. SBL is highly optimized for data access and is loaded with over 100 application-ready routines and functions. Not only that, RAVE tools automatically generate *bi-directional, reusable* source code as you build apps on-screen. And to work out a bug, just click into our full-featured debugger.

While everybody's talking about multimedia, Superbase is making it happen. With our extensible image API, support for 15 image types, video players and MCI, you can add lots of spice to otherwise ordinary applications. There's also a royalty-free, multiuser runtime license (available separately), so you can share your dynamic applications with anyone.



Superbase can get into just about anything you're into. Our Superbase SQL Library (available separately) supports more back-end SQL servers than any other product. Plus, Superbase directly reads and writes data from dBase® files, and accesses Paradox® data.



Superbase version 2.0 recently won Windows Magazine's 1993 Win Award and was one of the recommended products in *Software Digest*'s recent evaluation of Multiuser Database Programs for Windows! Superbase is so advanced, it makes all other databases seem, well, prehistoric.

1993 ANNUAL WINDOWS 100 W MAGAZINE WIN AWARD

$99 Regularly $495

So if you want more life for your applications (and you), call now and order Superbase. It's a killer app!

Superbase version 2.0—The Relational Database With Visual Power.

Only $99. Call 1-800-336-8360, Op. 543.*

SPC SOFTWARE PUBLISHING CORPORATION

*Regularly priced at $495. Limited time competitive upgrade offer. Good while supplies last. † July 1993 *Software Digest* (vol. 10.7). *Software Digest* is a registered trademark of NSTL, Inc., a McGraw-Hill company. Superbase, the Superbase Selection Bar and the Software Publishing Corporation logo are registered trademarks of Software Publishing Corporation or related companies. Software Publishing Corporation acknowledges all trademarks and the rights of trademarks owned by the companies referred to herein.

Dr. Dobb's SOURCEBOOK

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

SPECIAL ISSUE

Sound, Video, Action!

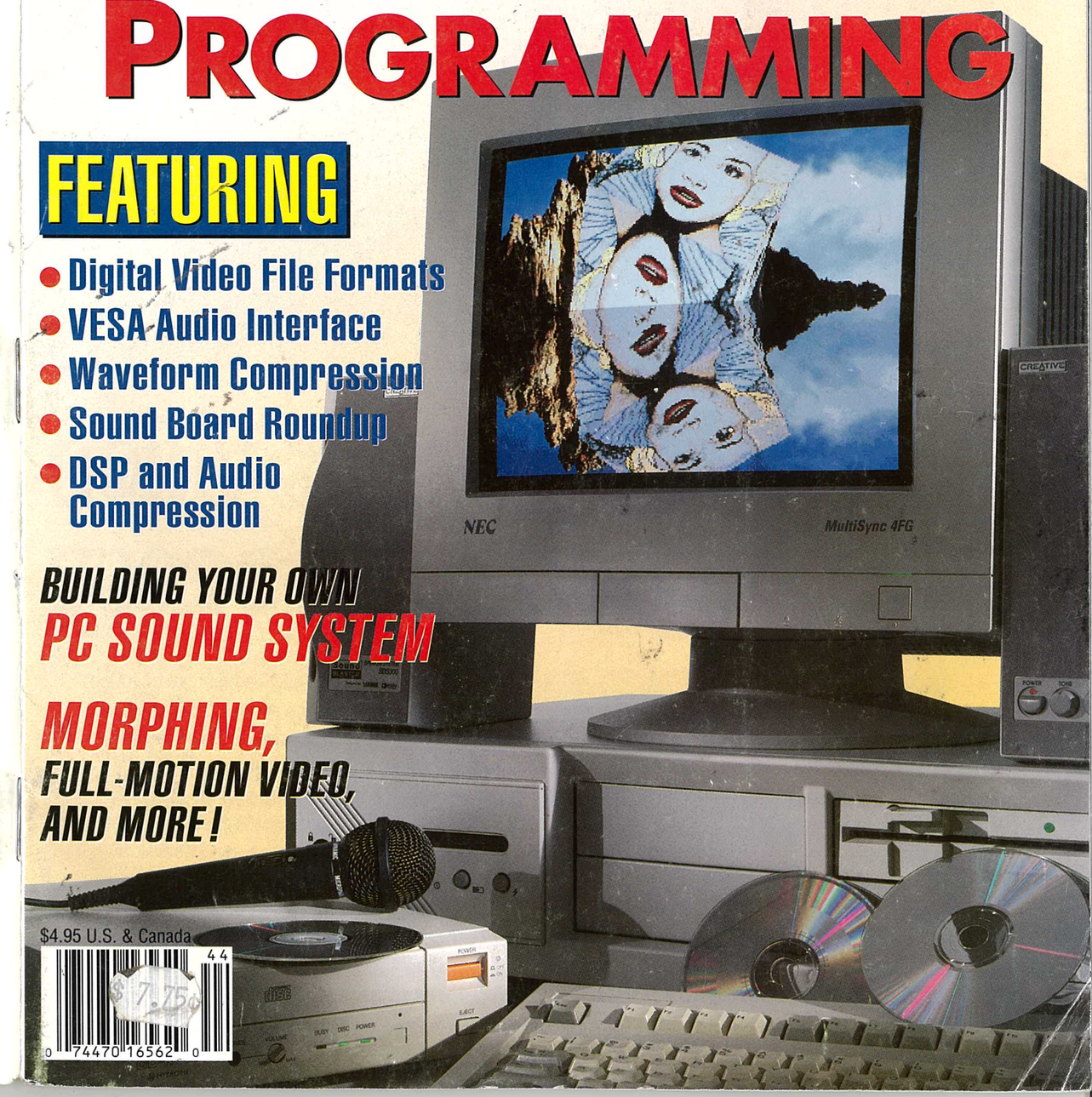
$ 7.75

TER 1994

MULTIMEDIA PROGRAMMING

FEATURING

- Digital Video File Formats
- VESA Audio Interface
- Waveform Compression
- Sound Board Roundup
- DSP and Audio Compression



BUILDING YOUR OWN PC SOUND SYSTEM

MORPHING, FULL-MOTION VIDEO, AND MORE!

$4.95 U.S. & Canada

0 74470 16562 0 44

$ 7.75